

An Evaluation of Live Sequence Charts and the Play-Engine

ALPER ERSOY



**KTH Computer Science
and Communication**

An Evaluation of Live Sequence Charts and the Play-Engine

A L P E R E R S O Y

Master's Thesis in Computer Science (20 credits)
at the Master of Science Programme in Engineering
and Management of Information Systems
Royal Institute of Technology year 2006
Supervisor at CSC was Karl Meinke
Examiner was Stefan Arnborg

TRITA-CSC-E 2006:037
ISRN-KTH/CSC/E--06/037--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se

An Evaluation of Live Sequence Charts and the Play-Engine

Abstract

Throughout the evolution of Software Engineering, one of the main struggles of software engineers has been that of capturing, specifying and managing software requirements so that they will lead to a good quality product that meets customers' needs, within limited time and budget. The requirements problem still remains unresolved. Studies of cost and frequency of the requirements errors underlines the need for validating requirements as early as possible. There are a number of mathematical specification languages and their tools that support automated and direct testing of specifications. However, they are not being accepted by the industry mainly because of their complexity. The widely accepted standard is specifying requirements in terms of use cases, each described by a set of sequence diagrams. The idea behind the Live Sequence Charts language and the Play-Engine is improving the system development setup by combining the expressive power of formal languages with the practical usage of the graphical specification languages.

Live Sequence Charts is a graphical requirements specification language that extends the popular UML Sequence Diagrams with more structure and some simple programming constructs. Keeping the similarity with the sequence diagrams, Live Sequence Charts, are used to capture and visualize the scenarios of behavior, however this time with a much more precise structure. So precise that it is executable. The Live Sequence Charts are sketched and executed with the tool called the Play-Engine.

In this thesis, we applied this new method for specifying and validating requirements in a project, together with some conventional techniques of the requirements activity. The method is then evaluated, considering our experiences with the project and some quality criteria for the method, the tool and the specification itself. The result of the evaluation implies that the technique are promising, but not yet mature enough to be used in industry, mainly because the tool is not stable yet. Among with the work considering the tool, we have addressed several other future works, with the vision of complementing these ideas and achieving their acceptance in the industry.

Acknowledgement

From a personal perspective, I should say that this work was rewarding. I hope the efforts might result in a contribution to the related knowledge body. Firstly, I would like to thank Professor Karl Meinke for providing this forward-looking research topic, letting me work on the project and being supportive and positive all through the project work. I would also like to thank Professor Stefan Arnborg for spending time to read and evaluate my thesis, and Ann Bengtson for helping with my registration and providing guidance on the process. I think I should also express my thanks to my uncle Bozkurt Ersoy, for helping with the informal user requirements and customer feedback that we used as example data in the project. At last, but not the least, I would like to thank to my family for providing constant moral support and motivation.

Table of Contents

1. Introduction.....	1
2. Problem.....	2
3. Related Work	4
3.1. Requirements Analysis Approaches	4
3.2. Specifying Requirements	4
4. Live Sequence Charts and the Play-Engine	7
4.1. Motivation.....	7
4.2. The Tool Play-Engine	8
4.3. Live Sequence Charts for System Development	10
4.4. Live Sequence Charts	13
5. Requirements Elicitation for a Slide Archive and Viewing Software.....	18
5.1. The Informal User Requirements Document.....	19
5.2. Actors.....	19
5.3. Use Cases	20
5.4. Played-in Behavior.....	22
6. Requirements Validation	38
6.1. Test Cases	38
6.2. Playing Out	42
7. Evaluation.....	50
7.1. Method and Tool.....	50
7.2. Specification	52
8. Conclusions.....	55
9. Future Work.....	56
References.....	58
Appendix A Questionnaire for Requirements Elicitation.....	61
Part I: Establishing the Customer or User Profile.....	61
Part II: Understanding the User Environment	61
Part III: Recap for Understanding.....	62
Part IV: Analyst's Inputs on Stakeholder's Problem (validate or invalidate assumptions)	63
Part V: Assessing the Opportunity.....	64
Part VI: Analyst's Summary	64

1. Introduction

Beginning from the first days of high level programming, software engineers and computer scientists have struggled with the concept of expressing requirements, that is, how to define our expectations from the system to be developed.

According to a large scale survey conducted by the European Software Process Improvement Training Initiative in 1995, largest problems in software development were found out to be requirements specification and managing customer requirements.

This report, which is written as a masters thesis in Nada, KTH, focuses on an emerging method for specifying requirements for reactive systems. The intended audience is master students studying computer science, information technology or software engineering and professionals interested in software requirements.

The report might be used as a guideline for practitioners who want to learn and employ this emerging methodology in software development projects. It might also be interesting for professionals who are acquainted with the language to see its application in a project.

The rest of the report is organized into following chapters:

- 2 Problem – Discussion of requirements errors and their costs
- 3 Related work – Presentation of the related work done both in the area of requirements analysis, and requirements specification
- 4 Live Sequence Charts and the Play-Engine – Discussion of the Live Sequence Charts language and the Play-Engine for modeling and describing user requirements
- 5 Requirements Elicitation for a Slide Archive and Viewing Software – Reflection of the experience gathered from using the underlined methodology in a project
- 6 Requirements Validation – Validating requirements with play-out
- 7 Evaluation – Evaluation of Live Sequence Charts language and the Play-Engine
- 8 Conclusions – Overall conclusion of the study
- 9 Further Work – Discussion of the future work that might be done considering the study

2. Problem

The main goal of software development is producing good quality software with limited time and funding, which meets customers' needs. One of the main struggles to reaching this goal is capturing, specifying and managing customer requirements, not only because the final software has to meet the customers' needs but also because requirements play a significant role on resources spent to develop software and the quality of the final product.

A study by the Standish Group in 1994 points out the top reasons for software development projects to be unsuccessful in terms of schedule and satisfying users' expectations. The top three reasons are found out to be: lack of user input with 13%, incomplete requirements and specifications with 12% and changing requirements and specifications with 12% of all projects examined. The main problems seem to be related with gathering, documenting and managing requirements.

Apart from the issues considering development problems, requirements also has a considerable impact on delivered product quality. Table 1, taken from [1], which summarizes a study by Casper Jones, 1994 [2], shows frequent sources of software defects, their potential to occur, removal efficiency and deliverance to the final product. It can be observed that almost one thirds of the defects delivered with the final product roots from requirement errors.

Table 1. Defect Summary

Defect Origins	Defect Potentials	Removal Efficiency	Delivered Defects
Requirements	1.00	77%	0.24
Design	1.26	85%	0.19
Coding	1.75	95%	0.09
Documentation	0.60	80%	0.12
Bad Fixes	0.40	70%	0.12
Total	5.00	85%	0.75

The cost to remove defects from the product is another important matter of concern. A study by Davis, 1993 [3], summarizes studies conducted by some major companies such as IBM and HP on the cost of errors on the different phases of development lifecycle. Table 2 below presents the effort required to detect and remove an error during the various stages of development.

Table 2. Relative cost of removing errors at different phases of development

Stage	Cost
Requirements Definition	0.1 – 0.2
Design	0.5
Coding	1
Unit Test	2
Acceptance Test	5

Maintenance	20
-------------	----

The errors to be corrected in each stage might be a result of activities performed on that stage or an undetected error from the preceding stages. It can easily be observed that the errors that have leaked from previous stages result in an overhead of costs, as the cost of correcting errors increases as we go through phases. One can also observe that the greatest overhead is caused by undetected requirements errors, as the cost to remove an error in the requirements phase is 5 to 10 times less than to remove it in the coding phase. Moreover undetected requirement errors might trigger design and coding errors, as the design and coding will be performed based on erroneous requirements. Thus, requirement errors should be detected as early as possible in a development lifecycle.

The requirements definition activity has to be improved to decrease costs, increase productivity, save time and deliver higher quality products. Pohl, 1993 [4] states the three dimensions of requirements engineering: representational issues which involve specifications, social domain issues which involve social process of eliciting requirements information and cognitive domain issues which are concerned with understanding the problem itself. In this report we will focus on representational issues on requirements problem and try to present the state of art and a recently proposed method in this dimension.

3. Related Work

3.1. *Requirements Analysis Approaches*

The two most widely used approaches to software requirements analysis are Structured Analysis and Object-oriented Analysis. Structured Analysis (SA) approach, which emerged in late 70's, is based on the functional-decomposition of the system and how data is processed by the system. Data flow diagrams are used to capture the functions needed by the system. In mid 80's, the approach was enhanced by adding the means to present state-based behavior of the system by the introduction of state diagrams and the language of statecharts. After decomposing the system into functions or activities, a statechart or state diagram is linked to each function or activity to illustrate its behavior. This approach has been widely used for real-time applications. Several SA methodologies and tools are available.

Object-oriented analysis (OOA), which emerged in the late 80's, concentrates on the things that make up the system, rather than system functions or activities. An object is an entity having a set of attributes and methods. Objects and the processing of data are used to capture, organize and communicate the problem domain of the system. Statecharts were often used to model behavior of object instances, just like in the SA approach. This approach covers software engineering concepts like encapsulation, abstraction, modularity and hierarchy. OOA supports reusability, extensibility and maintainability notions, which are needed in large systems development projects with long lifecycles. The efforts in this area have been finally compiled into a standard, the unified modeling language (UML) by the Object Management Group (OMG) in 1997.

3.2. *Specifying Requirements*

The purpose of depicting system behavior is to form a basis for the final implementation. However there exists another purpose as important as that, which is providing the means to see if the software fulfills its requirements, in earlier stages of development. The process of checking if the system fulfills its purpose is called verification and validation. The verification activity entails checking that the product matches with its specification, that is, its specified functional and non-functional requirements. Whereas validation activity is more abstract and general, as it intends to guarantee that the system is capable of performing customers' expectations. Although these two activities seem similar, the definition of the latter underlines that requirements specifications might not reflect the actual user expectations. Verification and validation activities are carried out in all stages of development starting with requirements reviews until code inspections and product testing. As we have mentioned before, the conflicts between user expectations and the specified requirements of the system should be resolved as early as possible since they will cost much more to be resolved in the later stages of development. Moreover, the defects

resulting from requirement errors are more likely to influence the final product quality.

The attempts to resolve conflicts between specified system requirements and actual expectations have lead to several techniques for specifying requirements, other than using natural language. These are:

Structured Natural Language is a manner of specifying requirements in which the requirements are specified in a standard structure. This type of notations makes use of some templates and the terminology to be used is often restricted. An example of such a template might include function, description, inputs, source, outputs, destination, action, requires, pre-condition, post-condition headings. Employing this kind of forms in requirements specifications eliminates the ambiguity problem with the natural language specifications to some extent, and usually maintains the understandability of requirements by the users. The main advantage of using this type of notations is that they help remove some problems of natural language with very little cost and effort for technical training. However the ambiguity problem still remains, especially in the case of complex computations. These notations also do not have standards and are adopted specifically in each project. Another drawback of such notations is that they have no tool support for validation or debugging. An early attempt to use structured natural language for specifying requirements was made by Heninger in 1980 [5], for an aircraft software system. A newer example of such a language can be found in [6], which is designed specifically for clocked computational tree logic.

Design Description Languages are similar to programming languages but they are more abstract in structure. This approach aims to define an operational model of the system. These languages are useful to define actions from a higher level, however they are inadequate to define detailed algorithms. There also exist tools to support these languages to check syntax and identify contradictions between requirements. The main drawback about using these languages is that they do not completely cover the implementation language but must be translated to it. So, most of the time, some details should be added to the specification while translating it to the implementation language which causes contradictions between the specification and the implementation. Moreover these languages require expensive technical training and are often not understood at all by users. Nowadays, this approach is very rarely used. Rapide prototyping language [7] can be given as an example of design description languages.

Mathematical Specifications or *Formal Specifications* use notations that are based on discrete mathematical constructs such as sets and finite state machines. They are extremely accurate in specifying requirements. The statements made in these languages can even be used directly in testing code correctness. Examples to these notations are Z [8], B [9], LOTOS [10] and VDM [11]. Although they have good tool support and are very powerful in expressing unambiguous requirements, most

customers do not understand them and do not want to accept as a contract. Moreover using these languages requires highly trained professionals.

Graphical Notations make use of graphical models and languages complemented with text footnotes to define functional requirements of the system. SADT language proposed by Schoman and Ross in 1977 [18] can be given as an early example of this approach. Other examples are SDL [19], Petri-nets [22] and UML [23]. Lately, using use case descriptions (Jacobson, 1993) [20] together with sequence diagrams (Stevens and Pooley, 1999) [21] has become the most popular method for specifying requirements. Sequence diagrams are used to describe the scenarios which are instances of the use cases. The relations between components of the system, and their relations with the environment, processes, messages, tasks that are contained in the use cases are all captured in the sequence charts. This technique also provides a nice abstraction for the system as use cases reflect the general requirements of the system and sequence diagrams provide the needed details for each use case. After the introduction of UML standard, this concept has been well studied and is being powerfully supported by a number of tools which enables the validation of these models. Although it still requires technical training, there exist many resources and courses to learn UML standard.

Among the other techniques for specifying requirements, graphical notations, UML in common has become popular due to its several advantages stated in [12]:

- Graphical representation makes easier to understand the system by different stake-holders
- Graphical representation provides a way to think about user requirements clearly
- Graphical representation provides a way for deriving test cases
- Graphical representation provides a way for identifying objects and system functions

In the following chapter, we will introduce LSCs which seems to be the next generation of the UML sequence diagrams graphical language to describe use-case requirements.

4. Live Sequence Charts and the Play-Engine

4.1. *Motivation*

As we are talking about specifying system behavior, we should first make clear the different types of behavior of a system. When we talk about the behavior of an object based system, we are talking about two types of behavior: inter-object and intra-object behavior. Inter-object behavior refers to the scenario based behavior of the system that covers the interactions and events between objects or components of the system to perform specific functionalities. This type of behavior is often employed in the early and late stages of development cycle, the requirements elicitation and testing. It is convenient to capture requirements in terms of scenarios because it is the most intuitive and natural way of stating the expectations from the system by users. Customers will often express the requirements in terms of some series of inputs to the system, what the system should do and how the operation should be carried out. On the other hand, intra-object behavior is the state based behavior of the system that covers the reaction of each object or component to the entire set of incoming events and changes that considers that object or component. This type of behavior is employed in the implementation phase, as we should provide each object or component with its complete reactivity to make the system execute.

Modeling the inter-object behavior of the system costs much less than the intra-object behavior in terms of time and funds, as the latter involves stating complex computations that should be performed by each object. The main intention of Live Sequence Charts and the Play-Engine is providing the means for monitoring scenario based behavior in operation and direct testing of requirements, before spending a lot of resources on intra-object state based models.

The currently popular approach to scenario based modeling, as we mentioned before, is using sequence diagrams, which originates from the language of message sequence charts, formerly accepted as a standard by International Telecommunication Union (ITU) [17]. The semantics of MSCs are meant to define the possible scenarios of system behavior, rather than the actual ones. They are composed of simple constraints on the partial order of possible events that might occur in a system run. Each object has a vertical line that denotes time, the higher events occur before the lower ones and each message is received after it is sent. MSCs can be used to specify the expected behavior of the system in requirements capturing stage and can act as test cases to check the correctness of the final system. However they are insufficient in defining the behavior of the system in a complete scenario based manner and providing the means for direct testing and debugging of requirements. Their expressiveness is limited because they lack variables, assignments, and constructs to express conditional information. Their modeling of real time is weak, as there is no means for specifying exact time between events. Live Sequence Charts language intends to take away these deficiencies of MSCs and provide the proper constructs to specify the behavior of systems precisely. Play-Engine provides the medium to sketch these diagrams (play-in) and see the behavior in operation (play-out).

The reason behind the deficiencies of sequence diagrams or MSCs is that they lack complete formal semantics. LSC language, on the contrary, provides a fully worked out formal semantics [14]. As we have mentioned before, formal specifications achieved precise expressiveness and has good tool support. However formal specifications have not been widely accepted by industry. The Workshop on Industrial-Strength Formal Specification Techniques (WITF) held in 1998 [13] discussed the issue of using formal specifications in industrial applications with both academic and professional participants. The main conclusion of the workshop was the need for “light-weight” formal methods and tools which do not require the analysts to learn and apply the theory and technology that the tools are based on. The main objective set by the participants was “we must make formal methods invisible and automatic.” The tool Play-Engine, developed by Rami Marelly and David Harel, at the Weizmann Institute of Science, in Rehovot, Israel, enables the analysts to perform system specification by directly interacting with the prototype GUI of the system and making use of the visual constructs of the language, which is referred to as playing in. Thus LSCs, together with their strong formal semantics and the tool Play-Engine with its “automatic” manner, seems to be fitting into the description of future formal method tools defined in the workshop.

Another remark made on the workshop was that the formal specification techniques should keep pace with the existing standards because the industry seems to rely on such standards due to their available support by vendors. The participants mentioned about the UML standard and stated that some techniques and tools should be integrated into UML to support automatic code generation, simulation, analysis and test generation. The work by Jochen Klose and Bernd Westphal [15] demonstrates that LSC specifications can indeed be related to UML models.

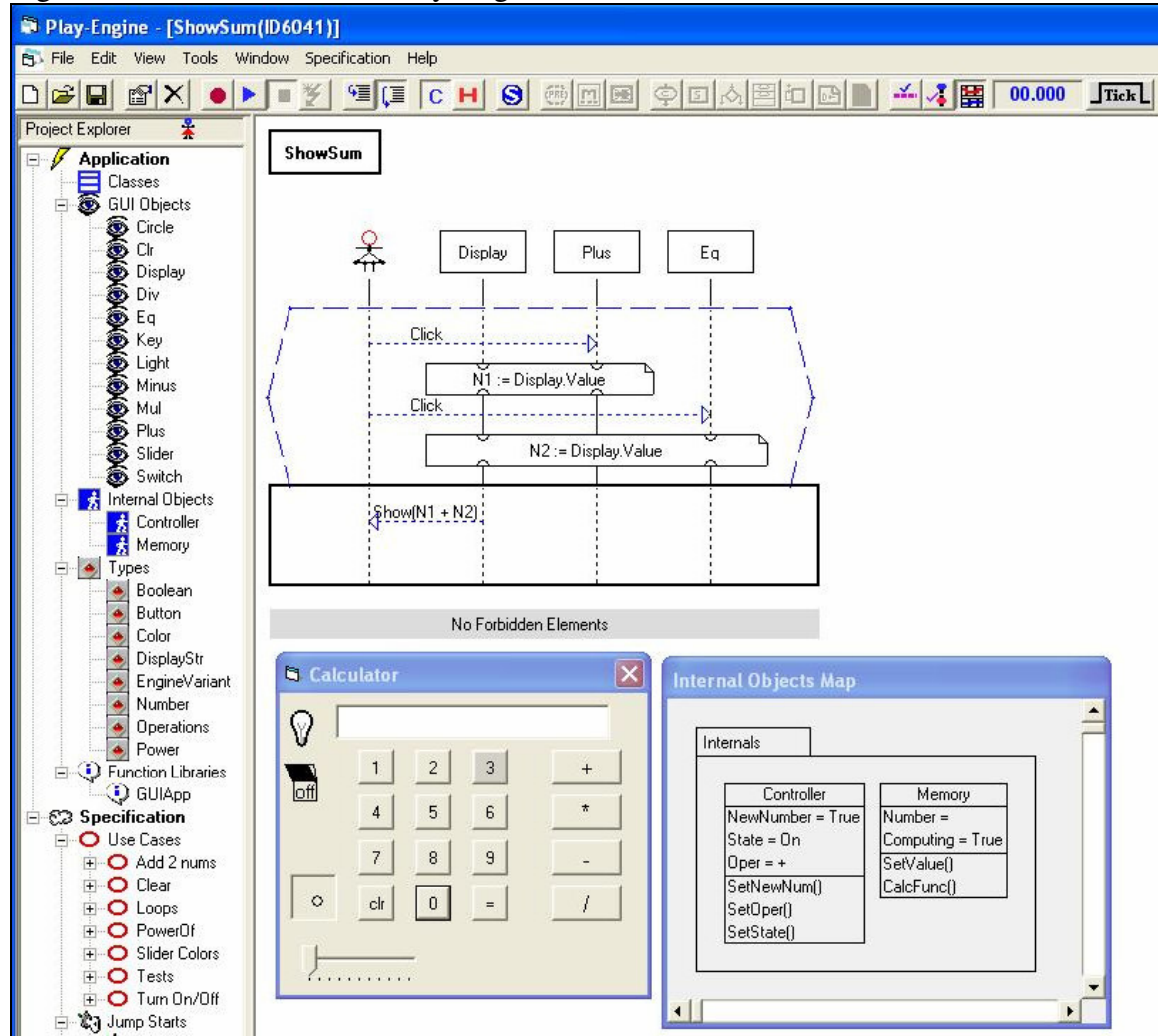
4.2. *The Tool Play-Engine*

Before going more in detail into the play-in/play-out approach and the LSC language, it is best give some insights into the tool in order to provide a general understanding of what we are talking about.

Figure 1 below is a screenshot taken from the Play-Engine. The record and play icons to the top are used to select the mode of operation, respectively playing in and playing out. The stop button is used when the user wants to switch between the modes. We can also see the graphical user interface of the system in the picture, this interface is a dll file built with Visual Basic 6.0, containing the necessary code that enables the Play-Engine to recognize and interact the GUI objects of the interface. The code for the recognition of the interface by the Play-Engine, together with the xml description of the properties for the GUI components, is generated with an add-on to the Visual Basic 6.0 called GUIEdit. Once in the play-in mode, the user can sketch LSCs by clicking and entering inputs into the user interface, and the internal objects map, which can also be seen in the screenshot. The Play-Engine automatically sketches the

events and messages performed by the user. The other icons to the right of record and play buttons that seem disabled in this picture are used to sketch various constructs into the language. When we need to include the properties of GUI objects or internal objects into those constructs, we again simply click and pick up the information we need directly from the user interface or the internal objects map. When in the play-out mode, the engine awaits user input and executes the specified behavior as a reaction to the entered inputs or performed events. The Play-Engine also visualizes the active LSCs and their states after each event.

Fig. 1. A screenshot from the Play-Engine



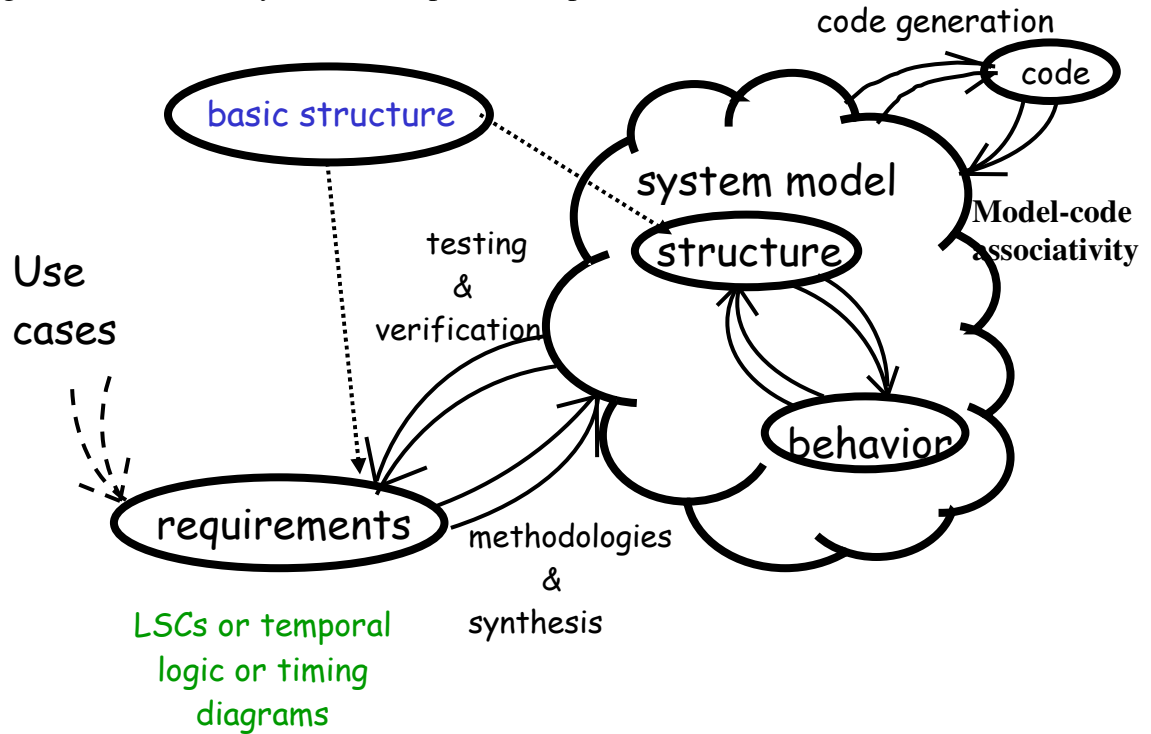
4.3. *Live Sequence Charts for System Development*

After having introduced the tool Play-Engine, now we will try to explain using LSC language and the Play-Engine as a development methodology, and its intended improvements to the conventional system development process.

The figure 2 shows the conventional setup for system development. Use-cases are transformed into formal requirements by system designers. This transformation is depicted with a dashed arrow because it is a soft methodological process. This requires nothing but a good understanding of the system and the target formal requirements language. A system model is tested and verified against these requirements written in a formal language. A nice technique to do this is producing sequence diagrams from the runs of the executable intra-object model. These diagrams produced by an automated tool can then be checked against the diagrams in the requirements specification. Although this is a powerful technique, it still has the drawbacks of classical testing and debugging. A system might have infinite number of runs, however our model checking covers only a limited number of those. So there will be some runs that have not been checked against requirements and some of those can be the ones that violate the requirements. On the other hand, LSC language and the Play-Engine tool provide us a computerized way of verifying if the model will satisfy the requirements. The highly expressive mechanisms and constructs of the language enable us to make sure that the system behaves as required, that is, the things that must not happen will not happen, and the things that must happen will happen.

The other activity involving requirements is the evolution of requirements into a system model. There exist a number of well studied methodologies dealing with this activity, offering guidelines, frameworks and step-by-step directions. Although these methodologies are useful, they are informal recommendations about the process. The play-in/play-out approach makes it possible to generate an implementation model directly from the requirements.

Fig. 2. Conventional system development setup

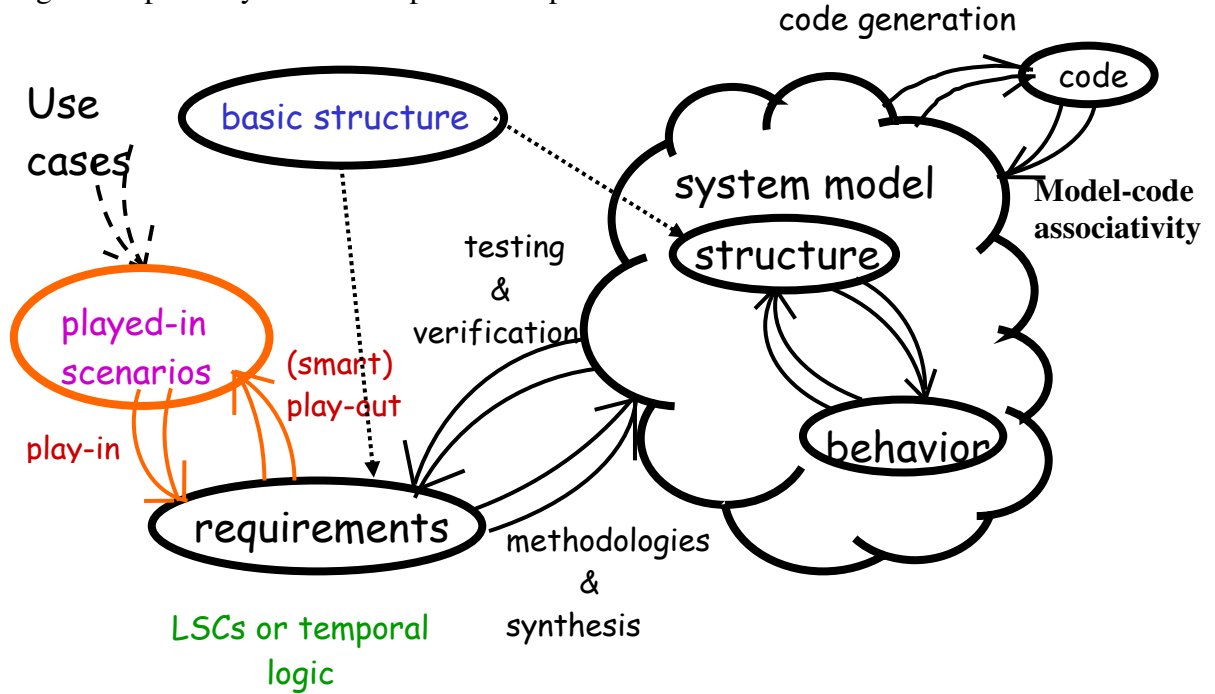


The play-in/play-out approach aims to complete the conventional development cycle by filling the gap between use cases and the formal requirement specification languages. The Play-Engine is intended to provide an automated way of specifying requirements with a natural and intuitive way that might address a wide variety of users. Figure 3 shows the proposed system development setup. In the proposed development cycle, requirements are specified in a high level of abstraction, without being have to work with a formal language. Firstly, a GUI of the system is built together with some definitions for each GUI object so that the Play-Engine can recognize them together with their attributes and methods. This GUI is then opened in the Play-Engine. Scenarios considering each use case are played in directly through the graphical user interface by entering inputs and performing events. The various constructs of the Play-Engine are also accessed through some icons to constrain these scenarios. Internal objects can be added to the system in Play-Engine and are accessed through an object map. As the user performs events by clicking on the objects either on the GUI or the object map, Play-Engine monitors the current status of the GUI and sketches the corresponding LSC.

The natural activity to be done right after specifying requirements is checking them against users` expectations. Play-out mechanism provides the means for testing and validating the specified behavior of the system from its inter-object model. In this mechanism, user simply executes the GUI of the system by clicking play-out mode, then enters inputs and performs events on the GUI as well as the internal objects. The scenario performed on the system might be a random one as well as a played in scenario. The Play-Engine keeps track of the actions and activates the systems

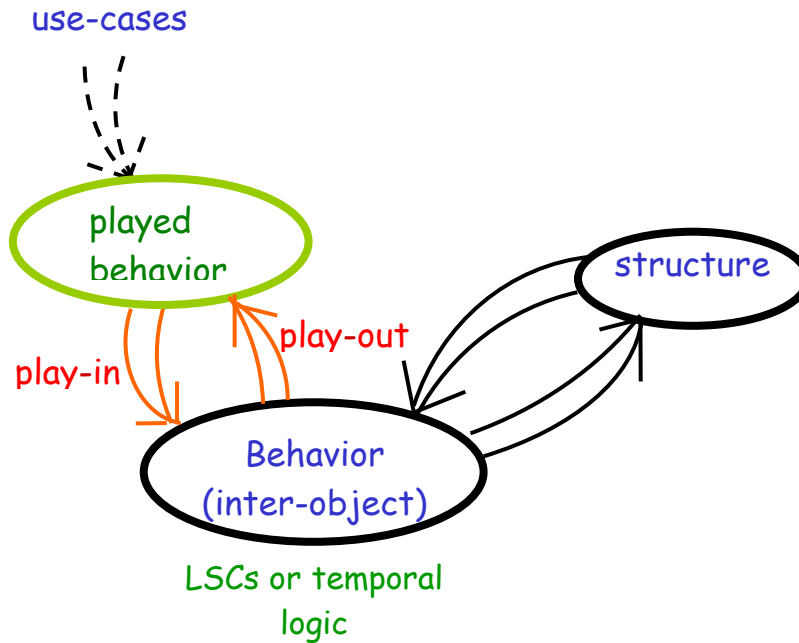
reaction according to them. The play-out functionality of the Play-Engine provides the means for working with an executable model, without having to build an intra-object model.

Fig. 3. Proposed system development setup



Harel and Marely [24] points out that, if these ideas are thoroughly worked out in the future, the behavior specified by playing in scenarios might be the only behavior that is to be specified for implementation of some systems. They also add that intra-object behavior will still be needed for complex systems but it will be greatly supported by inter-object behavior. Figure 4 shows this futuristic cycle for system development.

Fig. 4. A futuristic system development cycle



4.4. *Live Sequence Charts*

In order to address the deficiencies of MSCs, Harel and Damm extended the language of MSCs to become Live Sequence Charts in 1998. The term “live” was inspired from the charts capability of specifying the things that must occur.

Live Sequence Charts can be either universal or existential. Universal charts are enclosed within solid borderlines, whereas existential charts are enclosed within dashed borderlines. Existential charts are much similar to MSCs in structure and they only need one successful run to be satisfied. Existential charts are often used as system tests or examples of object interactions. Universal charts, on the other hand, are used in scenario based behavior specification that should be satisfied in all possible system runs at all times. A universal chart is composed of two parts: the prechart and the main chart. The prechart part contains the scenario that activates the LSC. When activated, the LSC will enter the main chart, which contains the scenario that is the reaction to the given scenario in the prechart. All together, a universal LSC can be seen as an if-then construct. A group of LSCs make up a set of action-reaction pairs among which the universal ones should be satisfied at any time interval in a system run.

A chart contains hot and cold elements. Hot elements, which are denoted with red solid lines, refer to the actions that must take place or the conditions that must be satisfied for system correctness. Cold elements, which are denoted with blue dashed lines, refer to the actions that might take place or the conditions that might be satisfied. We often use cold elements for control structures.

Live Sequence Charts language and the Play-Engine offers a number of mechanisms and constructs that are based on programming concepts to enable the precise, program-like specification of system behavior. These are:

Variables and Symbolic Messages

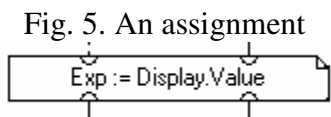
Denoting messages from user to object and in between objects can be done in two ways: symbolic mode and exact mode. The message labels in LSC language are of the form <Property Prefix> <Property Name> (<Value>). While in exact mode, the value to be passed in the message should be specified while denoting the message. However in symbolic mode, a variable can be passed in the message instead of an actual value. A message denoted in this manner means that it can take any value of the variables type. Same variable can be used more than once in a chart, which means that the same value should also be used in the other occurrences of the variable in a specific run. A variable is local to the chart that contains it, so it can be referenced only in that chart.

Implemented Functions

The modeler might want to include some data manipulation into the scenarios by means of some functions. However the syntax of the LSC language does not allow defining such functions. To address this need, the Play-Engine allows the GUI applications to supply their own implemented functions, each recognized by a name, names and types of parameters and the type of the return value. These functions can also be contained in function libraries and loaded into the Play-Engine. When the function should be applied, the Play-Engine queries the GUI application or the function library to obtain the return value for the given input values.

Assignments

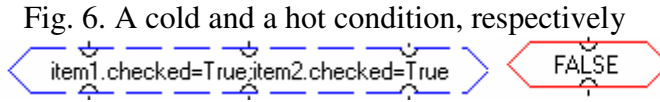
Assignment construct allows the user to save object properties or results of functions into variables. The assigned variable can then be used later in the LSC when the value is needed. The left side of the expression contains the assigned variable, and the right side contains a constant, a reference to an object property or a function result. Figure 5 shows the graphical representation of an assignment.



Conditions

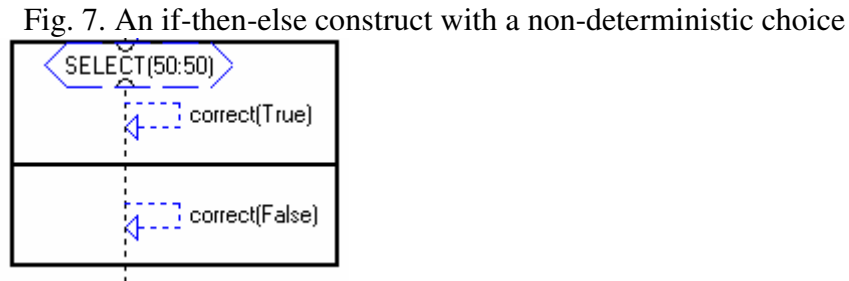
LSC language allows the users to specify conditions of two types: cold conditions and hot conditions, having different impacts on execution. A condition hexagon is stretched along several objects and connected to the instance lines of the referred objects. Even if there is no reference to an object in the condition, we might want to connect the object to the condition in order to synchronize it with the condition. Any object connected can not progress beyond the condition in its instance line until all of the connected objects reach the condition and the condition is evaluated. A cold condition, when evaluated false, causes its surrounding chart to be exited. However,

when a hot condition is evaluated false, it means that the requirements are violated, so it causes a system abort. Cold conditions are denoted with blue dashed hexagons and hot conditions are denoted with red solid hexagons. Figure 6 shows the graphical representation of cold and hot conditions.



Branching and Subcharts

Conditions can be used to control the flow of execution of a chart. If-then-else construct provided by the LSC language runs different scenario fragments depending on a condition. It is composed of two subcharts and a conditional on the top of the first subchart. First subchart, namely 'then', executes when the condition holds and second subchart, namely 'else', executes when the condition is false. Moreover if we use a 'SELECT' statement, a built in function of the Play-Engine to produce true or false values with a given probability specified by arguments, in the condition part of this construct then we can specify a non-deterministic choice. The use of subcharts is not only restricted to if-then-else constructs. They can be denoted without any condition. The purpose of sketching a subchart is making sure that all the instances participating in the subchart arrives at the subchart start before execution and subchart end before exiting. Figure 7 shows the graphical representation of an if-then-else construct with two subcharts, specifying a non-deterministic choice.



Loops

In some applications, specific actions are needed to be performed a number of times. The need for modeling such actions is covered by the loop construct of the LSC language. There are three kinds of loops defined in the language. A fixed loop iterates for a fixed number of times, which might be a constant value or a variable from the chart. An unbounded loop iterates until a cold condition inside it is evaluated as false. How many times a dynamic loop will iterate is specified by the user in run time and it is marked with a question sign. An example of a loop construct can be seen in figure 8.

Fig. 8. An unbounded loop



Time

Reactive systems often refer to time and react according to it. LSC language and the Play-Engine are also equipped with the constructs to express time. Play-Engine uses a discrete clock which is linked to the internal clock of the computer in use. The constructs for time are assignments and conditions. A combination of assignments and conditions are used to specify timing constraints. A vertical delay interval specifies the maximum and minimum allowed delays between two successive events. To define such an interval: time is stored right after the first event, a hot condition specifying minimal delay is placed just before the second event and a hot condition specifying maximal delay is placed right after the second event. Figure 9 shows the graphical representations of time assignment and condition.

Fig. 9. A time assignment and a time condition, respectively



Forbidden Elements

We might want to specify the things that must not occur as well as the things that might or must occur. One can forbid scenarios using the mechanisms described above, by placing the entire scenario in the prechart and a hot false condition in the main chart. Besides this technique, LSC language contains a more direct way of prohibiting behavior, called forbidden elements. These are forbidden messages and forbidden conditions. Forbidden messages are messages which are not permitted to take place in a given span. They might have four different levels of generality, which can be chosen inside the Play-Engine. We can choose the “specified message only” option to prohibit just the message we denote, if it is an exact message, and prohibit all the messages with the same name, if it is a symbolic message. The other levels of generality are “all messages between these objects” to prohibit all messages between the specified objects and “all messages” to prohibit all messages between all objects in the LSC. Forbidden conditions are conditions which are not allowed to hold anywhere in their given scope. Both forbidden messages and forbidden conditions might be hot or cold. The graphical representations of forbidden messages and conditions can be seen in figure 11 and figure 10 below.

Fig. 10. Forbidden conditions

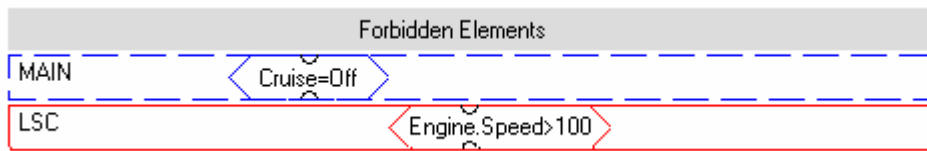
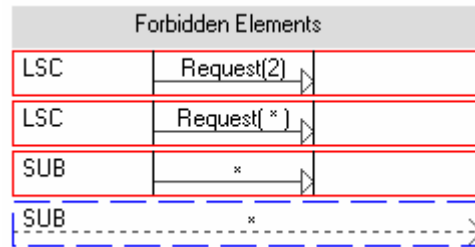


Fig. 11. Forbidden messages



5. Requirements Elicitation for a Slide Archive and Viewing Software

After having described the various features of the LSC language and the Play-Engine, now we will try to demonstrate the usage of these features in an application.

The system to be developed was a slide archive and viewing software. We have started the requirements elicitation, like in most of the projects, with an informal user requirements document written in natural language. We have then parsed this text and extracted actors and use cases to produce a use case model of the system. Then, according to this model, we have built a candidate GUI of the system and sent this to the customer, together with a survey (Appendix A) that basically follows the generic interview format given in [1]. The prototype GUI can be seen in figure 12 below. After receiving the feedback from customer, the use cases and the scenarios considering each of them were reviewed and modified. After having understood the basic behavior of the system, we have formalized each use case with a number of LSCs.

Fig. 12. The prototype GUI of the system

The screenshot shows a window titled "slide archiving software". Inside the window, there are several input fields and buttons. The fields are arranged in a grid-like fashion. The first row has three fields: "name", "founder", and "architect". The second row has four fields: "location", "architecture type", "date", and "period". The third row has a single field labeled "path" and a "browse" button next to it. Below the "path" field are two buttons: "index" and "list". To the left of the "description" field are three small square checkboxes. The "description" field is a large text area. Below the "description" field are four buttons: "print selected", "view selected", "view description", and "add description". At the bottom of the window is a section labeled "progress information" with a text field inside it.

5.1. The Informal User Requirements Document

Before going into more detail about the system, we will characterize the system from the customer's point of view by presenting the informal user requirements document:

The main use of the software to be developed is archiving pictures. The picture to be archived might be the photograph of an architectural construct such as a mosque, Turkish bath, house, palace or a drawing.

The software should provide the means for putting the pictures together as an album. While archiving each picture keywords will be given by the user. When a keyword is entered by the user, the pictures related with that keyword should be listed. For example, when the keyword "Istanbul" is entered all the constructs in Istanbul should be listed. After the list is obtained if we enter the keyword "palace" then the software should list all the palaces located in Istanbul. Or when we enter the name of the architectural construct, all the pictures considering that construct should be listed. The time period information might also be appended to each record so that when we enter a period for querying, all the pictures of the architectural constructs within the period will be listed.

As a summary, one or more keywords will be given to each picture and after forming the album, these pictures will be listed and viewed by means of the keywords entered by the user.

The software to be developed should also let the user to enter a short textual description of the picture so that when a picture is being shown, s/he can write the description looking at the picture and print out the picture when needed.

Finally the user should be able to carry the archive formed by the software and load it in another machine with the software installed and be able to do all the mentioned functionalities.

5.2. Actors

In order to provide a general understanding of the system, we should first define the actors participating in the system:

User: Someone who uses the slide archiving and viewing program to archive pictures, retrieve them with keywords, attach descriptions to them, view them and their descriptions, print them out and their descriptions.

5.3. Use Cases

Use cases have been designed considering the actor, his/her demands from the system, and his/her responsibilities. The text use case descriptions given below are composed of 4 parts: actors participating in the use case, pre-condition, post-condition and use case description. Pre-condition is the required system state to start the use case. Post-condition is the system state that should be obtained as a result of the use case.

Index picture

Actors: User

Precondition: There is at least one picture to be archived in the memory. Textboxes for entering the attributes for pictures are all empty.

Postcondition: All text boxes are set back to their initial state. “New record added” message is displayed in a message window, or status bar.

Description: User enters the name of the picture and specifies the path of the picture in memory by writing it to the path textbox. These two attributes are compulsory to be entered in order to be able to index the file. Other relevant attributes can also be entered to their relevant textboxes. Finally the user clicks the archive button. If the entered path is correct, system saves all the information entered in the database as a new record.

List pictures using keywords

Actors: User

Precondition: There is at least one picture in the current archive, textboxes for entering keywords are empty, text field for the name of the picture is empty.

Postcondition: The result of the query is listed in the list box, all the textboxes are set back to their initial states.

Description: User enters keywords to the textboxes and clicks the list button. At least one keyword should have been entered. System retrieves names of all the files having these attributes from the database and shows them in the list box.

Attach description

Actors: User

Precondition: There is at least one picture listed in the text area, that is, the user has performed a search in the archive. Text field for description is empty.

Postcondition: Text field for description is empty.

Description: User selects a picture from the list, writes the short description in the text area, and clicks the attach button. Only one item should be selected from the list. System appends this text to the record considering this picture in the database and displays “description attached” message in a message window or status bar.

View description

Actors: User

Precondition: There is at least one picture listed in the text area, that is, the user has performed a search in the archive. Text field for description is empty.

Postcondition: Text field for description contains the requested description.

Description: User selects a picture from the list and clicks the view button. Only one item should be selected from the list. The description is retrieved from the database and displayed inside description text field.

View selected pictures

Actors: User

Precondition: There is at least one picture listed in the text area, that is, the user has performed a search in the archive.

Postcondition: Pictures are being displayed in a new window.

Description: User selects at least one picture from the list and presses the view button. The system retrieves the memory location of the files from the database and sends them to the graphics display, the graphics display then displays them by opening from their memory locations.

Print out selected pictures

Actors: User

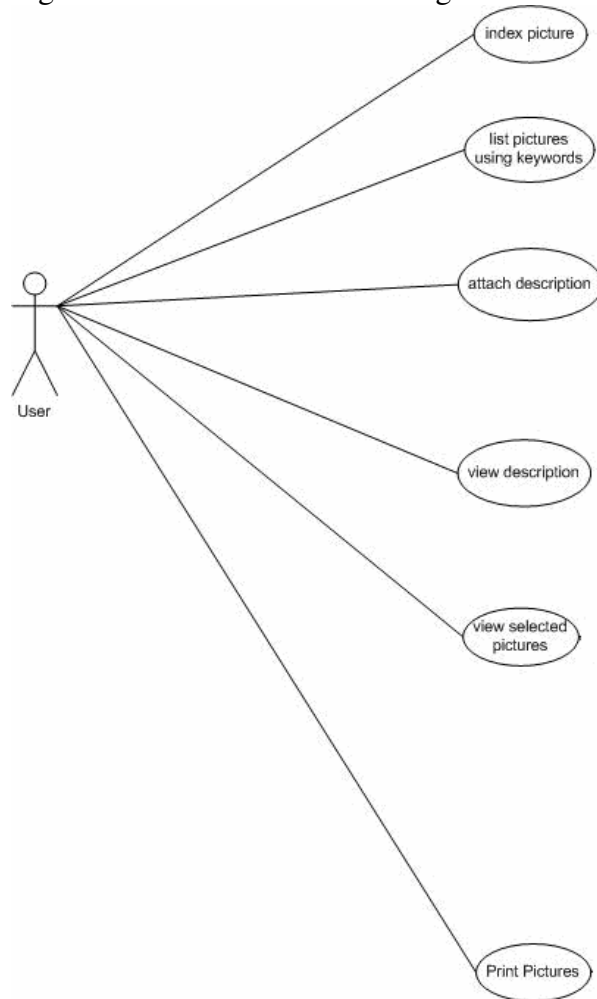
Precondition: There is at least one picture listed in the text area, that is, the user has performed a search in the archive.

Postcondition: Pictures are being printed by the printer. System displays a message indicating this.

Description: User selects at least one picture from the list and presses the print button. The system retrieves the memory location of the files from the database and sends it to the printer for printing.

Figure 13 below shows the use case diagram of the system, sketched in Microsoft Visio.

Fig. 13. Slide archive and viewing software use case diagram



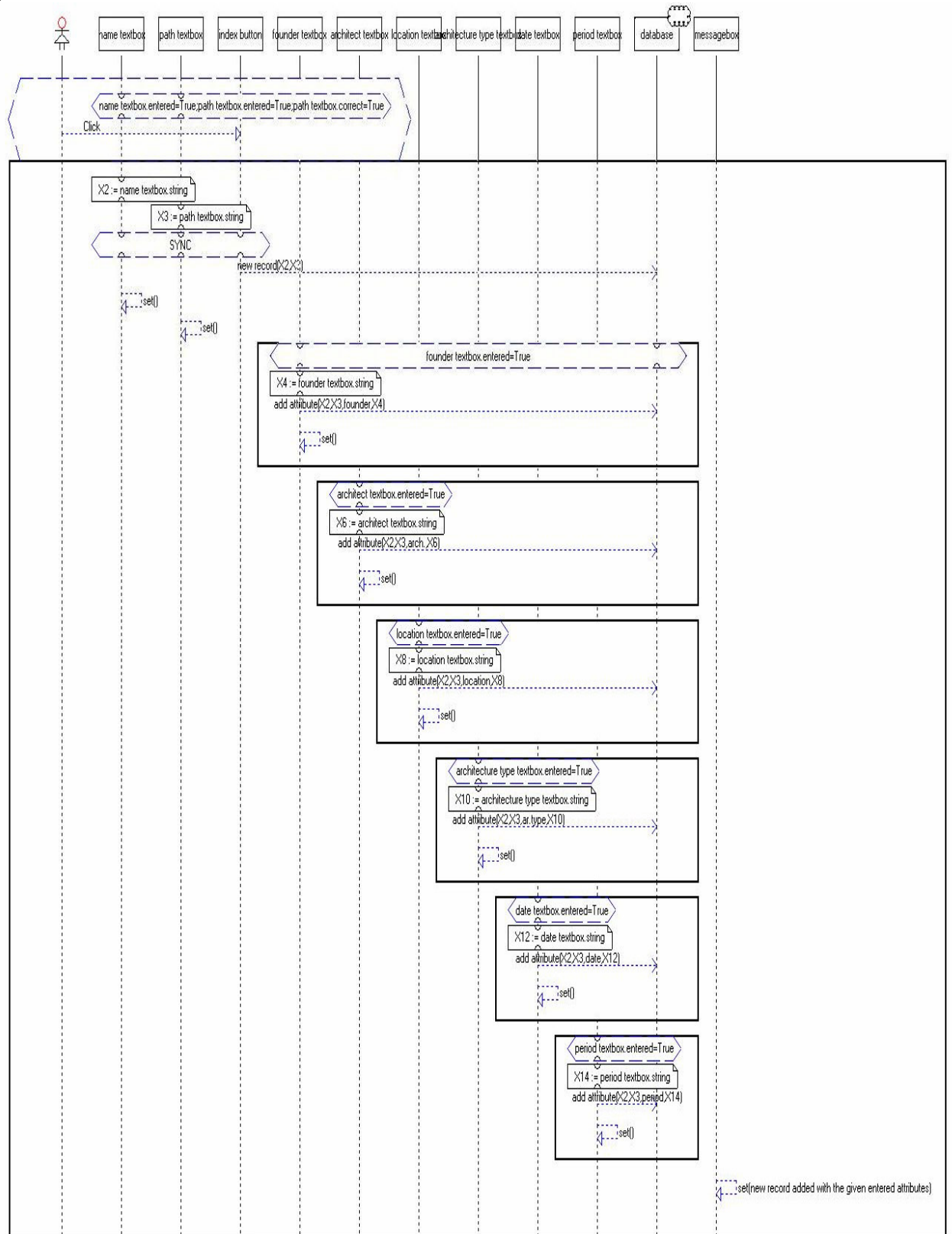
5.4. *Played-in Behavior*

After having described the actors and use cases, now we will describe the scenario based behavioral specifications we have played-in in order to formalize each use case. We will focus on each LSC and try to describe its structure and functioning. Guidelines on how to sketch various constructs and use mechanisms are given in more detail in the user manual [16].

Index Pictures

The following LSC, in figure 14, describes the scenario of indexing a file into the archive held in a database. We have assumed that the user must enter at least a name and a correct path for the memory location of the picture, in order to be able to append the picture to the archive. The cold condition at the beginning of the prechart checks if these conditions hold, by checking through the Boolean properties of the 'name textbox' and the 'path textbox' GUI objects. Once these conditions hold and the user clicks the index button, the LSC enters the main chart.

Fig. 14. Index file



No Forbidden Elements

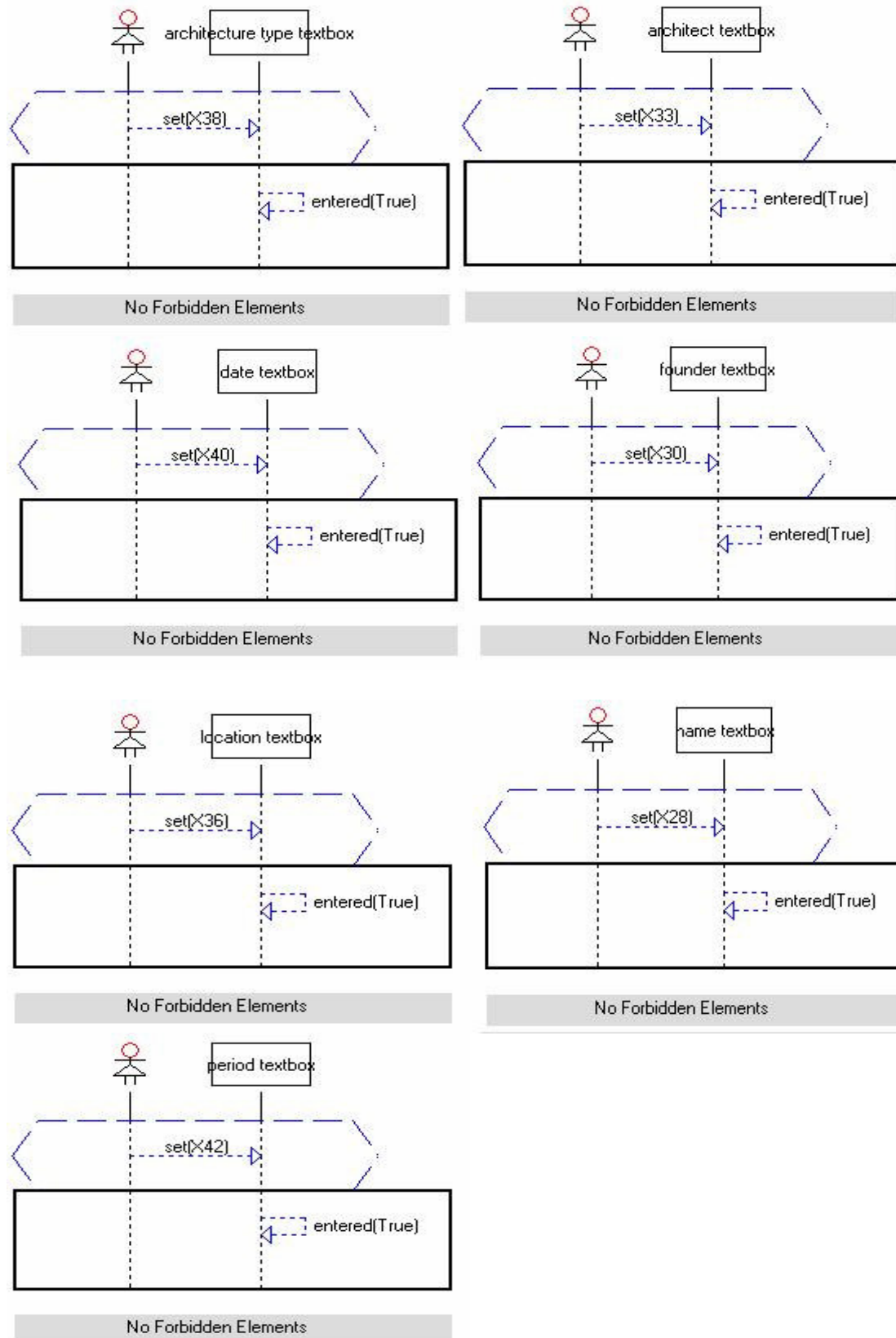
Appending a new record to the database is performed by making use of the *new record* function, which takes the ``name`` and the ``path`` of the picture as arguments, and creates a record in the database. In order to pass this information to the function, we first assign the values of the GUI objects' properties to variables. The GUI objects and the index button are synchronized using a true cold condition, which makes sure that each object reaches the end of their instance line up to the condition, before we try to create the record with the *new record* function. This way we make sure that the values are assigned to variables, before we pass them into the function. After creating the record with our function, we set the ``name textbox`` and the ``path textbox`` to their initial states as the information entered in them is no longer needed.

After having added the new record in the database, we then continue with adding the other attributes that has been entered by the user. Adding these values to a record is performed by the *add attribute* function. The *add attribute* function takes four parameters. The first two are the name and the path of the picture which are needed to identify the record (the row in the database) that the attribute is to be added. The third parameter states which column of that record will be used to store the new value passed in the fourth parameter. The attachment of attributes is performed by a series of if-else constructs with empty else parts. The cold conditions at the beginning of each sub-chart check if there is a value entered in specific GUI objects by looking at the values of the Boolean properties of each GUI object. If there is an entered value, it is first assigned to a variable and then added to the record with *add attribute* function by passing in the column name. After going through all the GUI objects in this manner, we display a message in the ``message box`` informing that the record has been added with all given attributes.

As we have said before, each text field GUI object has a Boolean property that states whether or not a value is entered in them. These Boolean properties are initially set to false and modified to true each time a value is entered in a text field. LSCs defining this reaction are given below in figure 15. As the entered values can be anything, the user events in the precharts are played in as symbolic.

The ``path textbox`` GUI object has an additional property from the others, which states whether or not the value entered in text field is correct. In the LSC in figure 16, after setting the ``entered`` property to true, we use an if-else construct with a non-deterministic choice to set the ``correct`` property. The *SELECT* function seen in the condition of the if-then-else construct is a built-in function of the Play-Engine and returns true or false with the probability values passed as arguments. In our case there is equal probability that the returned value is true or false. This way we simulate the correctness of a random user input.

Fig. 15. Enter picture properties



```

sequenceDiagram
    participant User
    participant PathTextbox as path textbox
    User->>PathTextbox: set(x44)
    activate PathTextbox
    PathTextbox->>PathTextbox: SELECT(50:50)
    PathTextbox->>PathTextbox: entered(True)
    PathTextbox->>PathTextbox: correct(True)
    PathTextbox->>PathTextbox: correct(False)
    deactivate PathTextbox
  
```

No Forbidden Elements

The system performs a query on the database making use of the entered values for listing and extracts the results. The specification on how to perform this is given in the *query database* LSC in figure 17. The minimum input needed to list items is a value in one of the textboxes. We decide if there is at least one input by checking through the Boolean ‘entered’ properties of each GUI object. However, because a logical or operation is not implemented in the language, we can not perform this check with a single condition. Instead, we have created a function that returns the logical or of two Boolean values entered as arguments, namely, *logicalor*. We use a series of assignments and a cold condition in the prechart in order to check through the properties. We take the logical or of all Boolean properties in pairs and assign each result to a variable. And then we perform further logical or operations with these values, and finally the logical or of all the properties is computed and assigned to a variable named ‘entered?’. This variable is then placed in a cold condition, which guards the main chart.

26

Boolean property ``hasitems`` of the `listbox` to true, which indicates that there are some items listed in the `listbox`. This information will be needed in some other use cases.

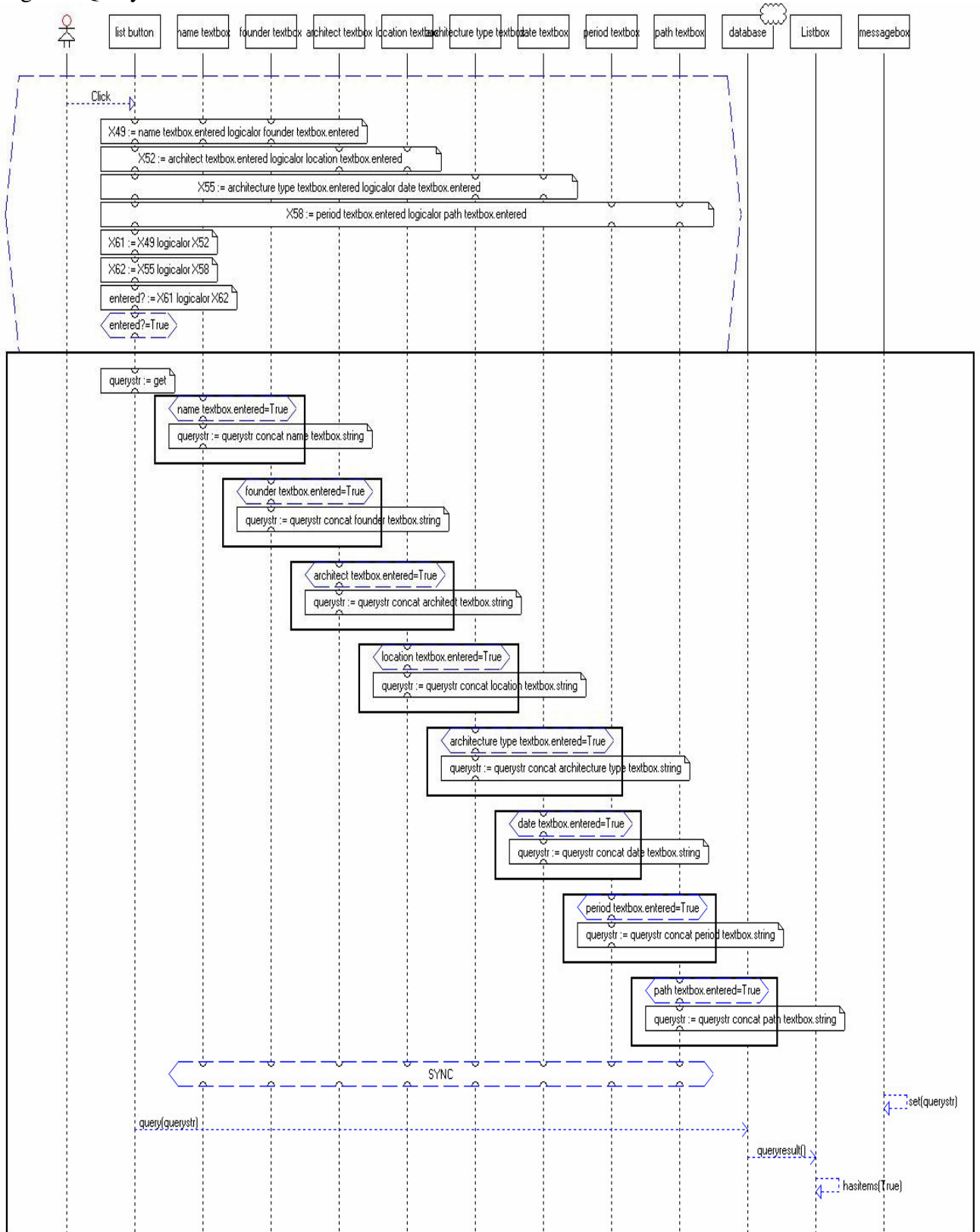
Concat and *logicalor* functions used in this LSC are user-defined functions and created using Visual Basic 6.0 together with the FuncEdit add-on and built into .dll files. These .dll files containing the functions are then included into the project in the Play-Engine. More information on how to create user defined function libraries can be found in the user manual [16].

Attach description

The specification of this use case has been handled a bit differently from the ones described before. The GUIEdit add-on in Visual Basic 6.0 enables us to define Play-Engine recognizable GUI objects and add properties to them. However, there is no means to define dynamic GUI objects created at runtime, such as items that will appear in the ``listbox`` where the results of a query will appear. In the *attach description* use case, user should have performed a search and selected an item from the ``listbox``, to which a textual description will be attached. To be able to specify this use case we have hardcoded that each time we click the ``list button``, there will be three items named `picture1`, `picture2` and `picture3`, listed in the ``listbox``. We have also placed three checkbox GUI objects to the left of the ``listbox``, to make the selection of items possible. These checkbox GUI objects are named, ``item1``, ``item2`` and ``item3``.

The LSC in figure 18 specifies the main scenario for attaching a description to a record. The entry condition to this use case is that there should be some items in the ``listbox``, the user should have selected one of those, and entered a description in the ``description textbox`` GUI object. We check the Boolean ``hasitems`` property of the ``listbox`` to determine if there are listed items, and the Boolean ``entered`` property of the ``description textbox`` to determine if there is an entered description, in a cold condition guarding the main chart. The Boolean ``entered`` property of the ``description textbox`` is initially false and modified as true each time some text is entered in this GUI object with the LSC in figure 19. We also perform a series of assignments and use the *logicalor* function, right after user clicks the ``add description button``, to determine if there is at least one item selected from the listed items and pass this information to the cold condition at the end of the prechart with a Boolean variable. If the user clicks the ``add description button`` and the condition is satisfied, the LSC in figure 18 enters its main chart. However, the user requirements imply that a description can be added to one picture at a time. In this case, as we are checking that there is at least, but not precisely one item selected, we might violate the requirements if the user has selected more than one item. In order to deal with this, we use a technique to catch these violations. We place the whole undesired scenario in the prechart and place a hot false condition in the main chart, so that the system is exited when a violation of requirements occurs. The LSCs in figure 20, figure 21, figure 22 and figure 23 covers the four possible unwanted scenarios of selecting more than one item and trying to attach a description. The system will be exited if any one of the scenarios in these LSCs occurs.

Fig. 17. Query database



No Forbidden Elements

Fig. 18. Add description

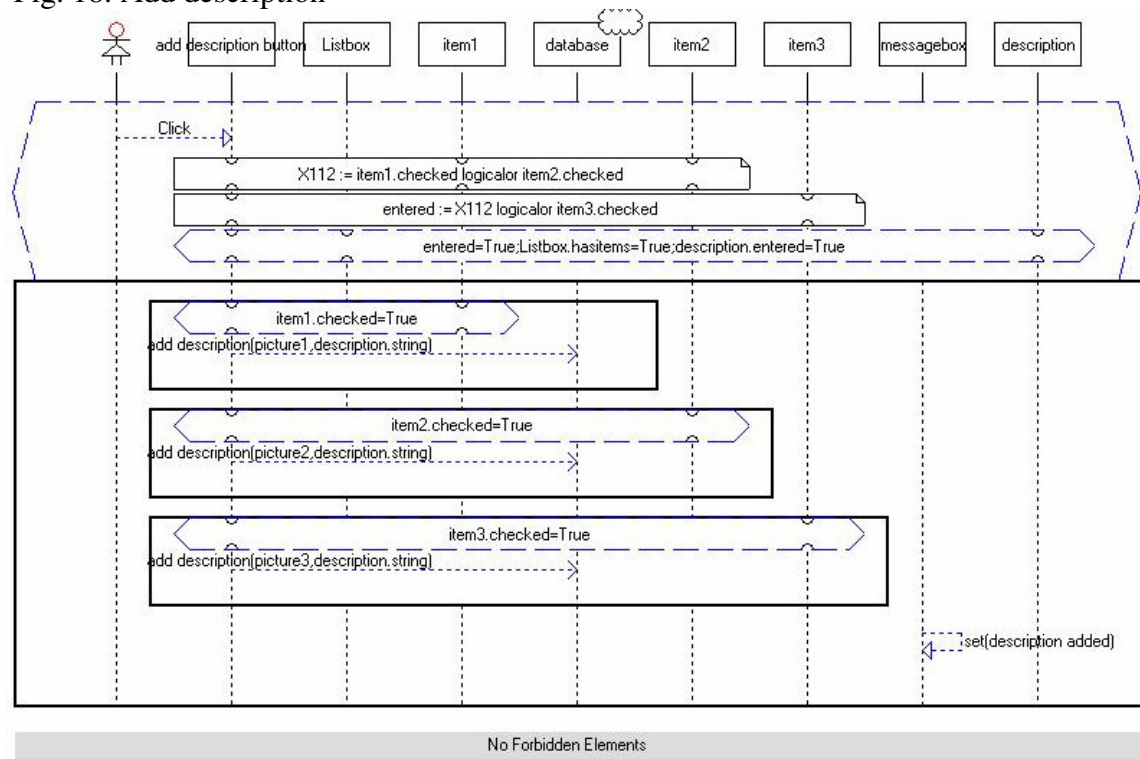


Fig. 19. Enter description

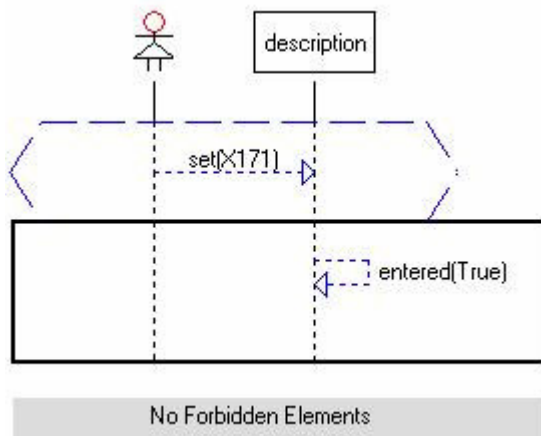


Fig. 20. Forbidden scenario 1

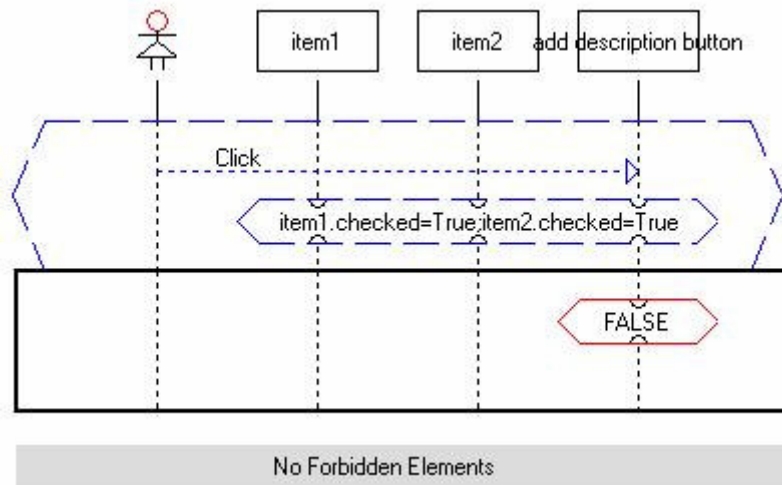


Fig. 21. Forbidden scenario 2

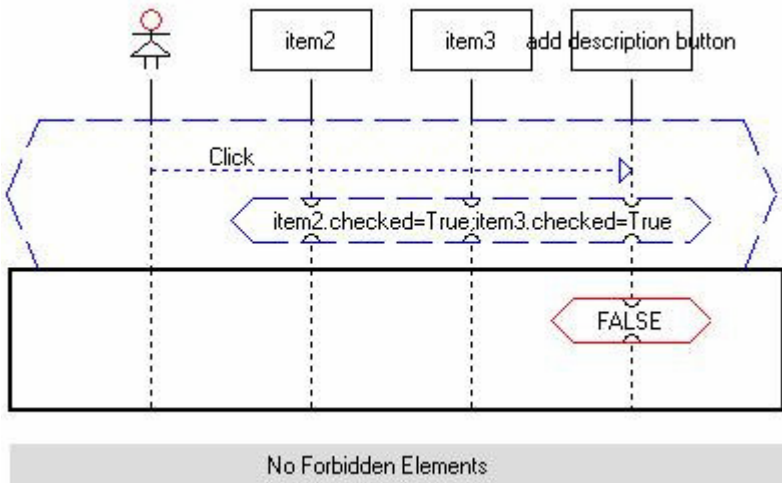


Fig. 22. Forbidden scenario 3

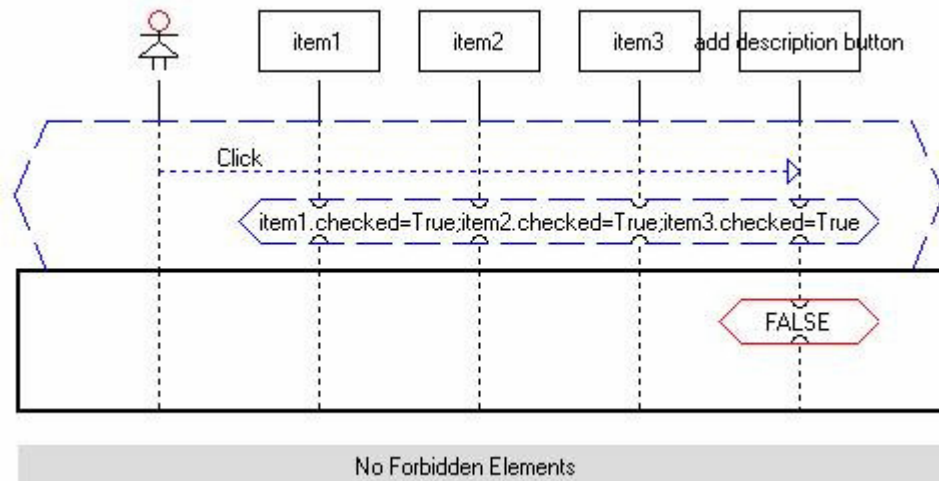
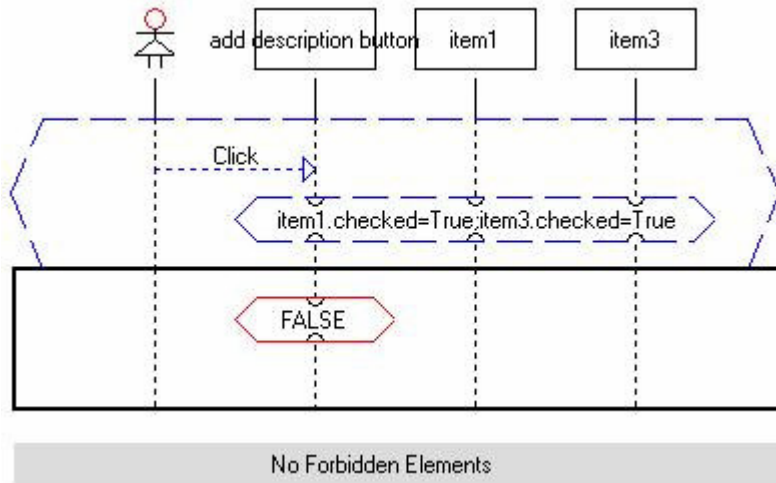


Fig. 23. Forbidden scenario 4



Once we are into the main chart of the LSC in figure 18, we find out which one of the three items have been selected with if constructs and add the description string in the textbox to that item with the *add description* function. This function takes the item and the string description as arguments. Finally a message is displayed to indicate that the description has been added.

View Description

The LSC in figure 24 specifies the main scenario for viewing the description of a record. The entry condition to this use case is that there should be some items in the 'listbox' and the user should have selected one of those. As the use case involves selecting items from the 'listbox', we will again make use of the hardcoded items and the checkboxes just like in the attach description use case. We check the Boolean 'hasitems' property of the 'listbox' to determine if there is listed items, in a cold condition guarding the main chart. We also perform a series of assignments and the *logicalor* function, right after the user clicks the 'view description button', to determine if there is at least one item selected from the listed items and pass this information to the cold condition at the end of the prechart. Just like the attach description use case, we want the descriptions to be viewed one at a time, so we again need the means for making sure that there is only one item selected from the 'listbox'. This situation is handled in the same manner as the attach description use case, with forbidden scenarios defined in the LSCs in figure 25, figure 26, figure 27, and figure 28, by placing the complete unwanted scenario in the prechart and a single false hot condition in the main chart.

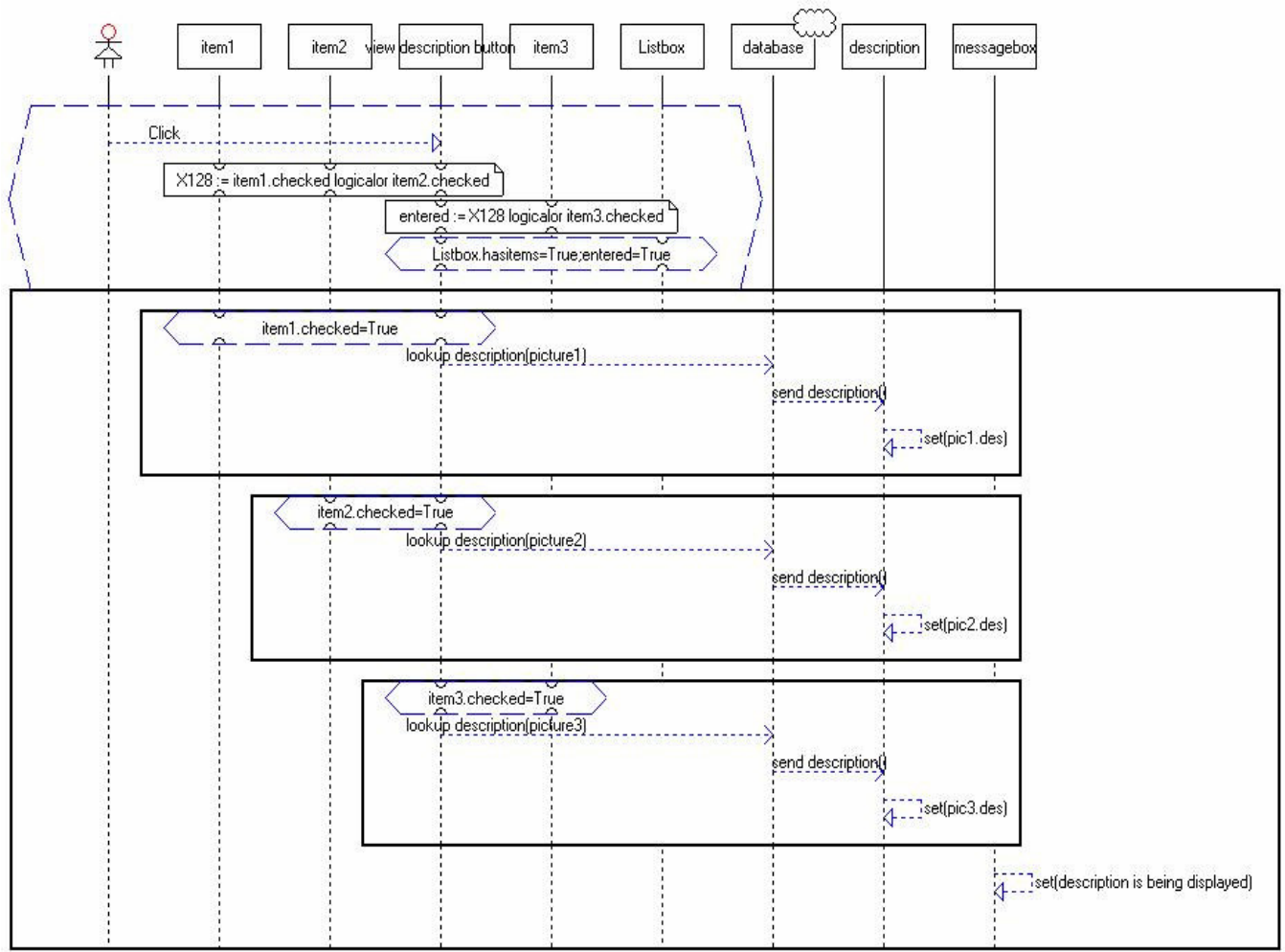
When we enter the main chart of the LSC in figure 24, we determine which item was selected by checking through the Boolean 'checked' properties of the checkbox GUI objects. Once we find out which item was selected, we pass this item to a function named *lookup description*, which will trigger a search in the database for the description of this item. The database then returns the result to the description textbox where it is supposed to be demonstrated. As this part is hardcoded, depending on which item is selected, we display a short text that stands for the description of the item at the description textbox.

Finally, we display a message at the 'message box' to indicate that the LSC terminated successfully and the description is being displayed.

View selected pictures

The LSC in figure 29 describes the scenario of viewing the pictures that has been selected by the user. The entry condition to this use case is that there should be some items in the 'listbox' and the user should have selected one of those. We carry out a series of assignments and use the *logicalor* function, right after user clicks the 'add description button', to determine if there is at least one item selected from the listed items and pass this information to a cold condition with the *entered* Boolean variable. These assignments simply perform a logical or operation on the Boolean 'checked' properties of the three items. These properties are initially set to false, and alternated between false and true values each time the user clicks on the items as specified in the LSCs in figure 30. We also check the Boolean 'hasitems' property of the 'listbox' to determine if there are listed items, in the cold condition guarding the main chart. As can be inferred from the description and the LSC, this use case also makes use of the hardcoded items.

Fig. 24. View description for the selected item



No Forbidden Elements

Fig. 25. Forbidden scenario 1

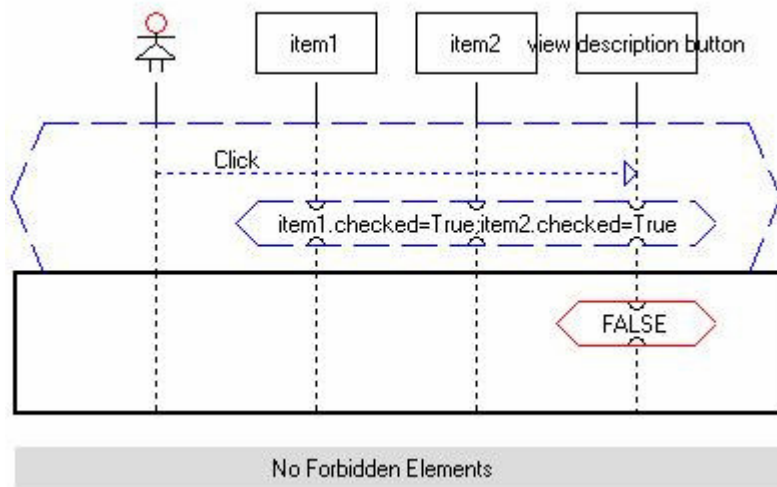


Fig. 26. Forbidden scenario 2

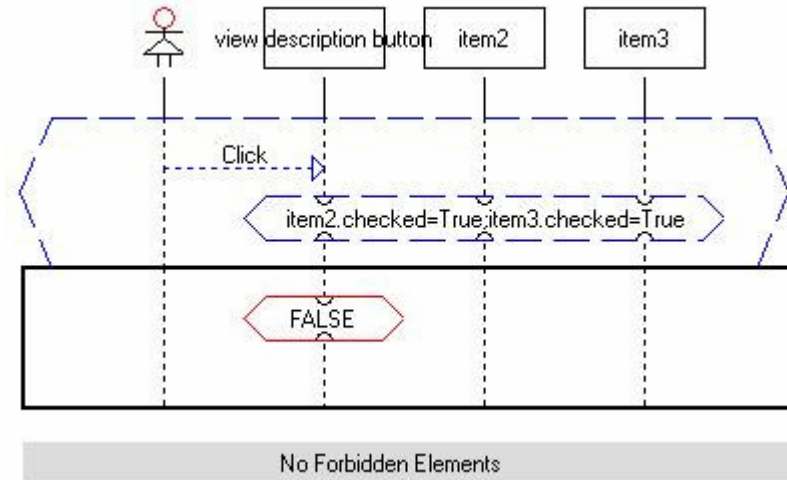


Fig. 27. Forbidden scenario 3

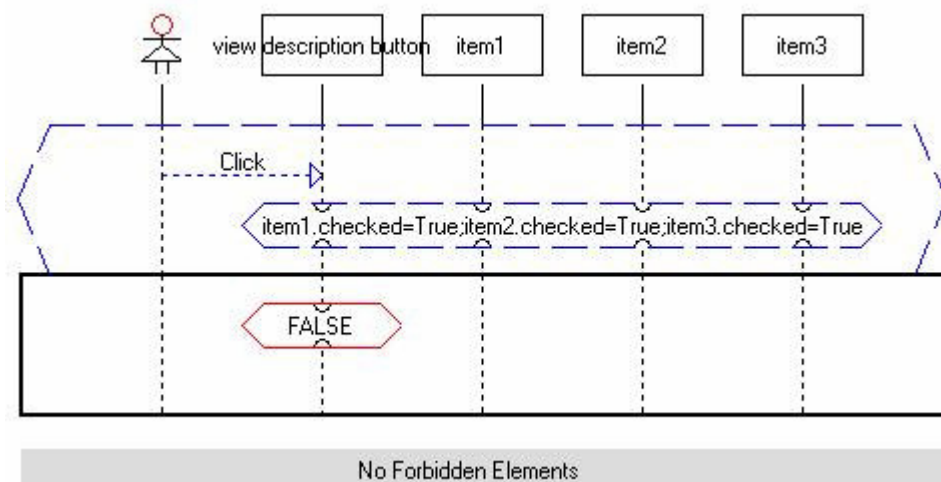


Fig. 28. Forbidden scenario 4

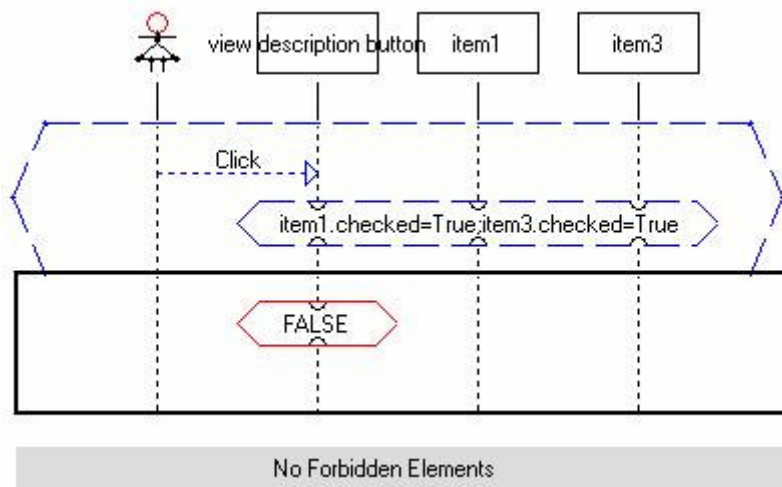


Fig. 29. View selected item

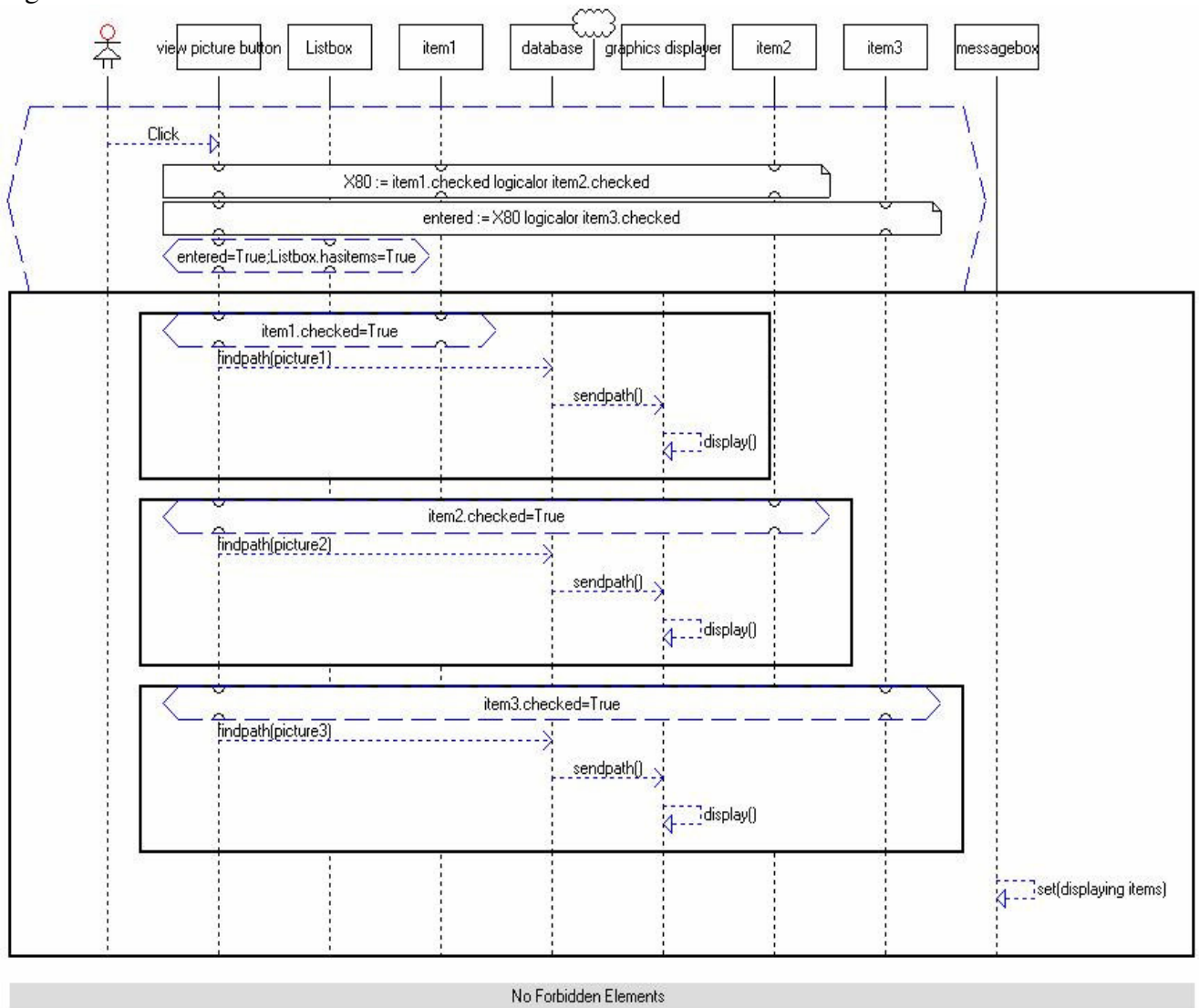
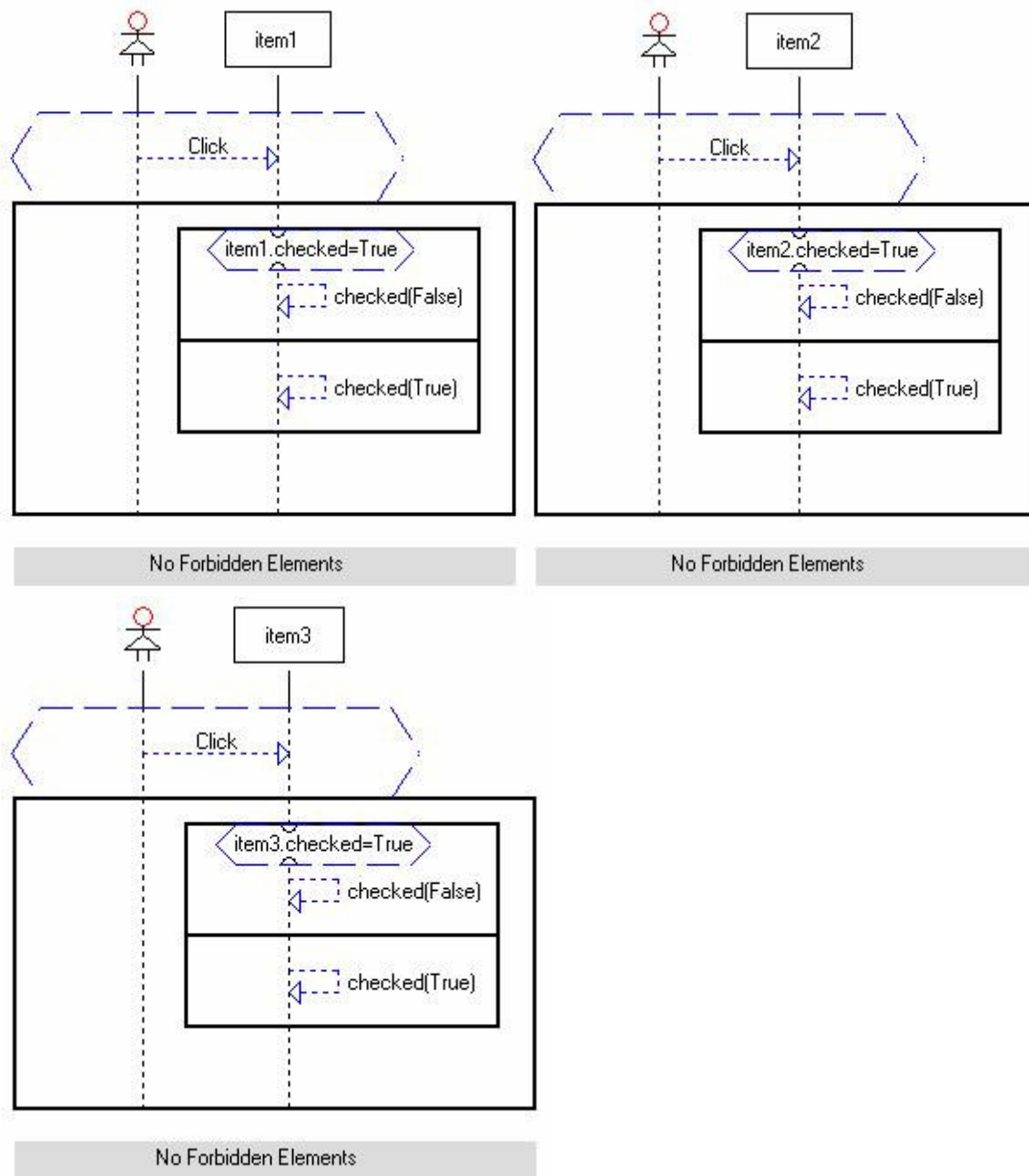


Fig. 30. Select items with checkboxes



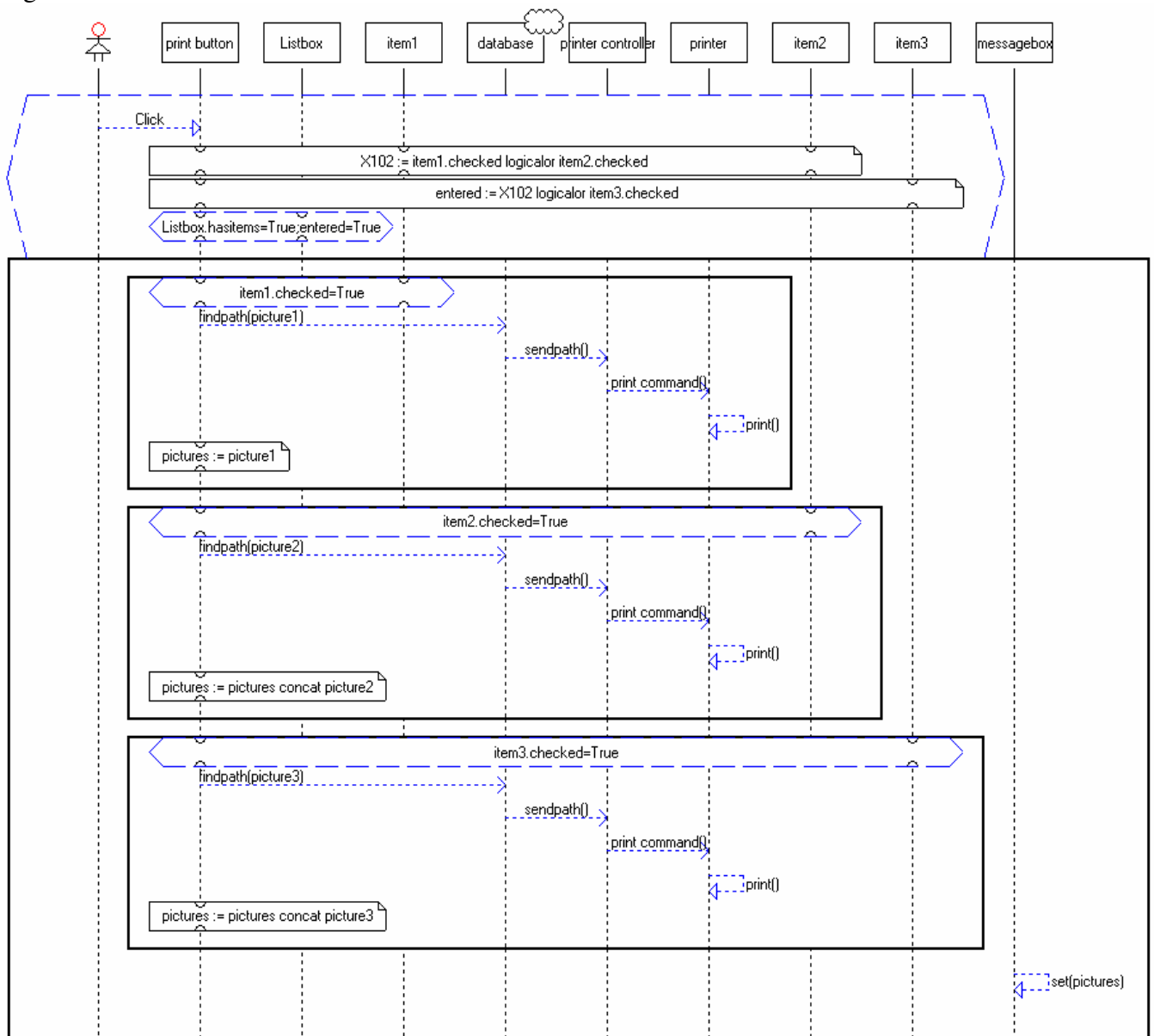
In the main chart, we go through each checkbox and find out if they are checked or not, using three if constructs. For each selected item, we pass that item to a function named *findpath*, which will trigger a search in the database for the path of this item. Once the path is sent to the 'graphics displayer' with the *sendpath* function, the graphics displayer executes the *display* function, which will open the picture from its location and display it on the screen.

Print out selected pictures

The LSC in figure 31 describes the scenario for printing out selected pictures. To be able to print one or more pictures, the user must have selected at least one item from the 'listbox'. The Boolean value indicating this is computed with two assignments and the *logical or* function, right after the user clicks the 'print button'. We then place this Boolean value together with the 'hasitems' property of the 'listbox' in a cold condition just before the main chart to check if the entry conditions are satisfied.

Once the preconditions of the scenario are satisfied, we enter the main chart. In the main chart we determine which items have been selected, making use of if structures which check the Boolean 'checked' properties of the checkboxes. For each selected item, we pass that item to the *findpath* function, which forces a search in the database for the path of this item. The path is then sent to the 'printer controller' internal object by the database, with the *sendpath* function. Then, having received the memory location of the file, the 'printer controller' sends the *print command* to the 'printer', which prints the pictures. After the picture has been printed, we append its name to a string variable. Right before the LSC terminates, we display this string in our 'messagebox', in order to demonstrate which pictures are being printed.

Fig. 31. Print selected



No Forbidden Elements

6. Requirements Validation

Now that we have specified the requirements in terms of use cases and formalized each use case with LSCs, it is time to make sure that our specification reflects the users' intentions. As mentioned before, the conventional way to do this goes through developing an intra-object model, or an executable prototype and using model execution for testing. Instead of following this track, we will perform the testing with our readily available inter-object behavioral specification, using play-out, the complementary mechanism to play-in.

In play-out mode, we are limited to end-user or external environment actions. As we perform these actions on the GUI and the internal objects, the Play-Engine monitors these actions interacting with the GUI application and causes the reactive actions to occur, as specified with the universal charts. This provides us a simulation of an executable application. The user interacting with the GUI to enter input, and the Play-Engine interacting with it to monitor the system state and present the output, is equivalent to working with an executable intra-object model to validate the requirements.

6.1. Test Cases

The system was tested for different combinations of inputs in play-out mode. In order to validate the requirements, we have produced some test cases based on each use case. A test case is defined as "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement." by IEEE standards [25]. Taking this definition as basis, we will now present our test cases considering each use case, stating the set of inputs and expected results. Each use case is tested for the basic flow and some alternate flows.

Index picture

The test cases generated for index file use case is given below in table 3. The "basic" test case is used to test the basic flow of the index pictures use case. The "alternate 1" and the "alternate 2" test cases are designated to test the two possible alternative flows of the use case, in which the minimum information to create a record (a name and a path) is not input by the user.

Table 3. Test cases for the index picture use case

Name	Test input description	Expected results
Basic	• User enters a `Name` • User enters a `Date` • User enters an `Architect Name` • User enters a `Path` • User clicks `Add Record` Button	• A new record is created in the database using the entered `name` and `path`. • The values of entered `date` and `architect name` are

		added to this record as attributes. • A message is displayed in the GUI stating that the record has been added with the given values.
Alternate 1	• User enters a `Date` • User enters an `Architect Name` • User enters a `Path` • User clicks `Add Record` Button	• An error message, stating that the record can not be created, is displayed.
Alternate 2	• User enters a `Name` • User enters a `Date` • User enters an `Architect Name` • User clicks `Add Record` Button	• An error message, stating that the record can not be created, is displayed.

List pictures using keywords

Table 4 below shows the test case generated for the list pictures use case, which is designated to test the basic flow of the use case. This test case should ideally be executed after we have executed the basic flow of index picture use case, so that there will be some items to be listed in the database.

Table 4. Test cases for the list pictures use case.

Name	Test input description	Expected results
Basic	• User enters a `Name` • User Enters an `Architect Name` • User Clicks the `List` Button	• A query string is created using the entered values. • The query is executed on the database and the results are shown in a listbox.

Attach description

The test cases generated for the basic and two alternative flows of the attach description use case are given in table 5 below. The basic flow of the list pictures use case should be executed before performing these test cases, so that there will be some items listed in our listbox and the user can select them. In this use case, a description is attached to one picture at a time, so, the user should select just one item from the listbox before attempting to attach a description. The “alternate 1” and “alternate 2” test cases covers the alternate flows of the use case, in which the user selects more than one item or no items to attach a description.

Table 5. Test cases for the attach description use case.

Name	Test input description	Expected results
Basic	• User selects an item from the listbox. • User Enters a Description. • User Clicks `Add Description`	• The entered value for description is added to the chosen record as an attribute. • A message indicating that the

	Button	description is added is displayed.
Alternate 1	• User selects an item from the listbox. • User selects another item from the listbox. • User Enters a Description. • User Clicks `Add Description` Button	• An error message, indicating that the description can not be added, is displayed.
Alternate 2	• User Enters a Description. • User Clicks `Add Description` Button	• An error message, indicating that the description can not be added, is displayed.

View description

The test cases generated for the view description use case are given in table 6 below. In this use case, the user should select one and only one item from the listbox in order to be able to view its description. The alternative flows of the use case, in which the user selects more than one item or no items, are covered with “alternate 1” and “alternate 2” test cases. The basic flow of the list pictures use case should be executed before performing these test cases.

Table 6. Test cases for the view description use case.

Name	Test input description	Expected results
Basic	• User selects an item from the listbox. • User Clicks `View Description` Button	• The system retrieves the description from the database and displays it in the GUI.
Alternate 1	• User selects an item from the listbox. • User selects another item from the listbox. • User Clicks `View Description` Button	• An error message, indicating that a description can not be viewed, is displayed.
Alternate 2	• User Clicks `View Description` Button	• An error message, indicating that a description can not be viewed, is displayed.

View selected pictures

In order to view pictures, user ideally selects one or more items from the list and clicks the `View Pictures` button. This basic flow of the use case is covered in the “basic” test case. The alternative flow, in which the user attempts to view pictures without selecting any item from the list, is covered in “alternate 1” test case. The basic flow of the list pictures use case should be executed before performing these test cases.

Table 7. Test cases for the view pictures use case.

Name	Test input description	Expected results
Basic	• User selects a number of items from the listbox. • User clicks the 'View Pictures' Button.	• The system retrieves the paths of the selected pictures from the database and displays them on the screen. • A message, indicating that the pictures are being shown, is displayed.
Alternate 1	• User clicks the 'View Pictures' Button.	• An error message, indicating that a picture can not be viewed, is displayed.

Print out selected pictures

In this use case, the user ideally selects one or more items from the list and clicks the 'Print' button. This basic flow of the use case is covered in the "basic" test case. The "alternate 1" test case is designated to test an alternative flow of this use case, in which the user tries to print pictures without selecting any items from the listbox. As these test cases make use of the listed items in the listbox, a successful execution of the basic flow of the list pictures use case should precede the execution these test cases.

Table 8. Test cases for the print pictures use case.

Name	Test input description	Expected results
Basic	• User selects a number of items from the listbox. • User clicks the 'Print' Button.	• The system retrieves the paths of the selected pictures from the database and executes a print command on the printer, which then prints the pictures. • A message, indicating that the pictures are being printed, is displayed.
Alternate 1	• User clicks the 'Print' Button.	• An error message, indicating that a picture can not be printed, is displayed.

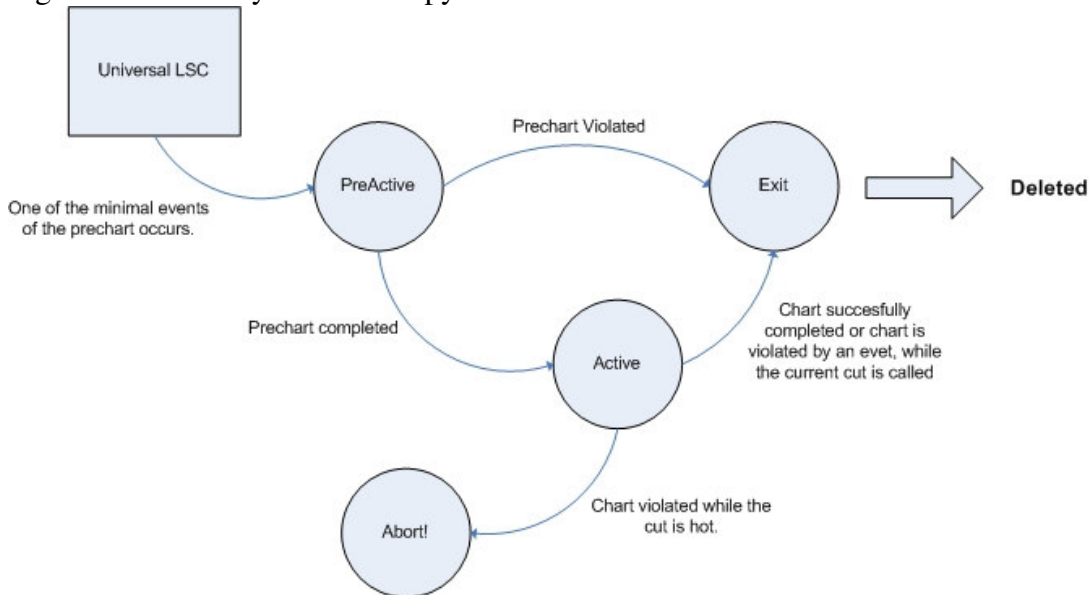
All of the test cases given above have been applied directly to our specification in play-out mode. The LSC states after each user event and the complete execution of these test scenarios have been captured and compiled into a web site, in order to demonstrate the validation of the requirements to the customer. The website can be accessed from the address: <http://www.nada.kth.se/projects/playengine/>.

6.2. Playing Out

After having described the test cases we have used in validation, now we will try to explain how these tests were carried out in play-out mode together with fundamental concepts of the play-out mechanism. We will present one of the performed play-out executions and try to illustrate play-out concepts on this run.

During play-out, a universal LSC becomes activated each time a minimal event that appears in its prechart occurs. After the LSC is activated, each of its instances begins to progress starting from their initial locations. The mapping of each instance in an LSC to its current location is called a *cut* of the LSC. If all the instances of a cut are in cold locations, then it is said to be a cold cut. If, at least one of the instances of a cut is in a hot location, then it is said to be a hot cut. When activated, a copy of the LSC is created and if the events occur in the order dictated by the LSC's prechart, the cut of the copy is propagated. The LSC is said to be in preactive mode when its cut resides in its prechart. Once all the events in the prechart occur, the cut progresses to the main chart and the LSC is said to be in active mode. If the cut successfully progresses through all the locations of the main chart as well, the LSC terminates and its copy is deleted. If the LSC is violated while it is in the preactive mode, it terminates again without causing an error, indicating that there is no reaction needed to be taken by the system. In case of a violation in the active mode, and while the LSC is in a cold cut, the LSC terminates without an error, as there is no obliged events to take place. However, if an LSC in the active mode is violated during a hot cut, that is, there is at least a hot event or a hot condition in the cut that has to be realized in the next step for the correctness of requirements, then the LSC aborts causing an error. Figure 32 below illustrates the basic lifecycle of the copy of a universal LSC.

Fig. 32. Basic lifecycle of the copy of a universal LSC.



Now we will go through the basic flow of view description use case in play-out mode to further explain the lifecycle of an LSC. The user selects the first item and clicks the 'view description' button. The event of clicking this button is looked up in all universal charts' precharts. This event is found in the LSCs 'view description', 'unwanted1', 'unwanted2', 'unwanted3', and 'unwanted4'. These charts become activated and a live copy of each is created in preactive mode. Figures 33, 34, 35, 36 and 37 show the LSCs in preactive mode, right after the user clicks the button. As there are no other events in the precharts of these LSCs, after the clicking event, Play-Engine starts performing the assignments and evaluating the conditions specified.

The prechart of 'view description' LSC, as we have described before, checks if the 'listbox' contains some items and there is at least one selected item with a cold condition that guards the main chart. As this condition is satisfied, the cut progresses to main chart and the copy of the LSC enters active mode (figure 38). Then Play-Engine goes on with evaluating the cold conditions in the precharts of the LSCs for unwanted scenarios. Each of these conditions checks if there is more than one items selected. As we have selected just 'item1', these conditions are all evaluated as false. LSCs terminate without causing an error because they are violated during a cold cut (figures 39, 40, 41 and 42).

Fig. 33. View description LSC in preactive mode, activated with a click event

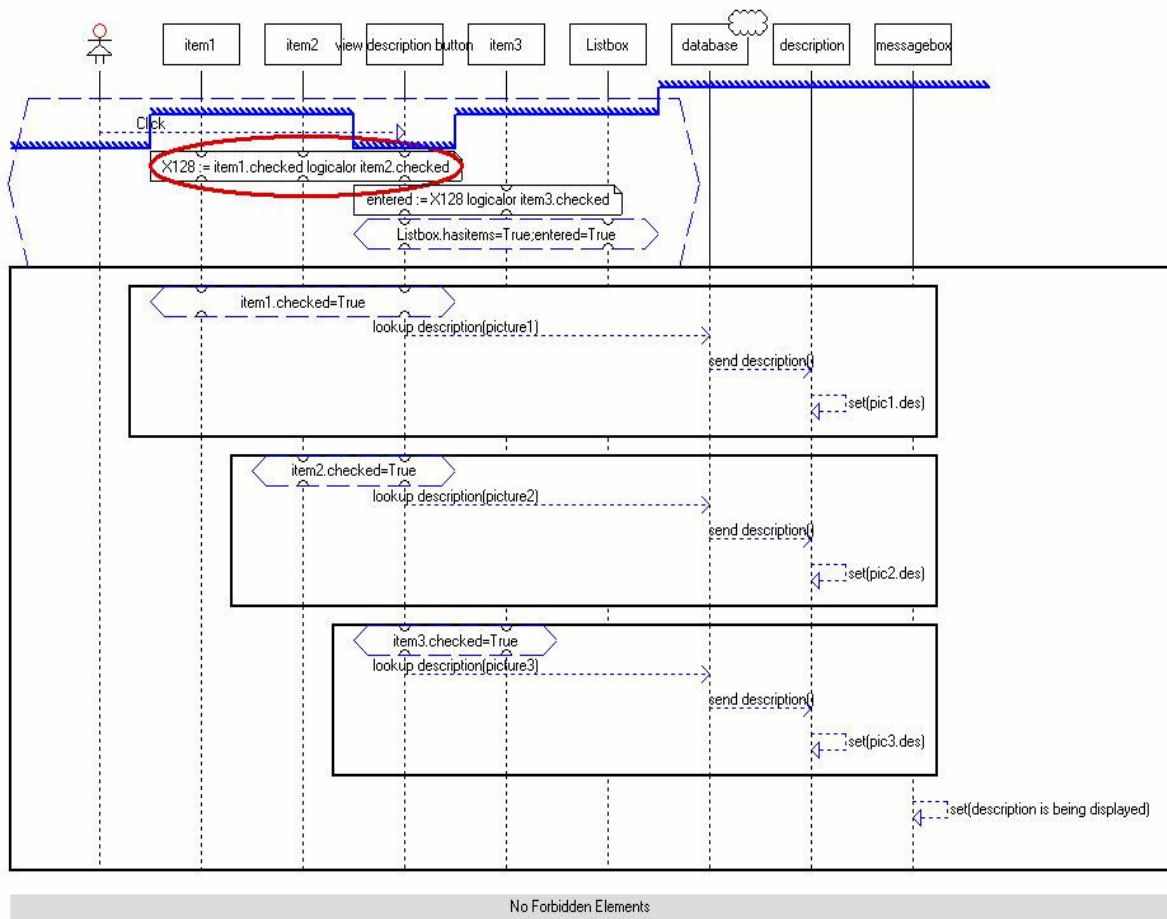


Fig. 34. Forbidden scenario 1 LSC in preactive mode, activated with a click event

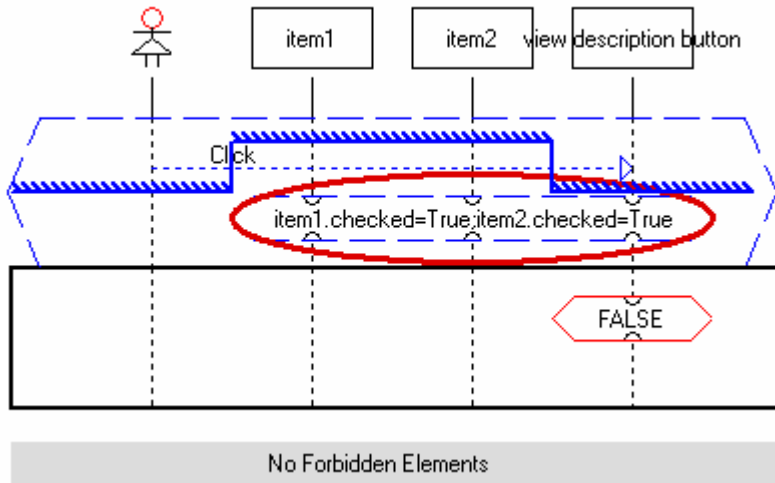


Fig. 35. Forbidden scenario 2 LSC in preactive mode, activated with a click event

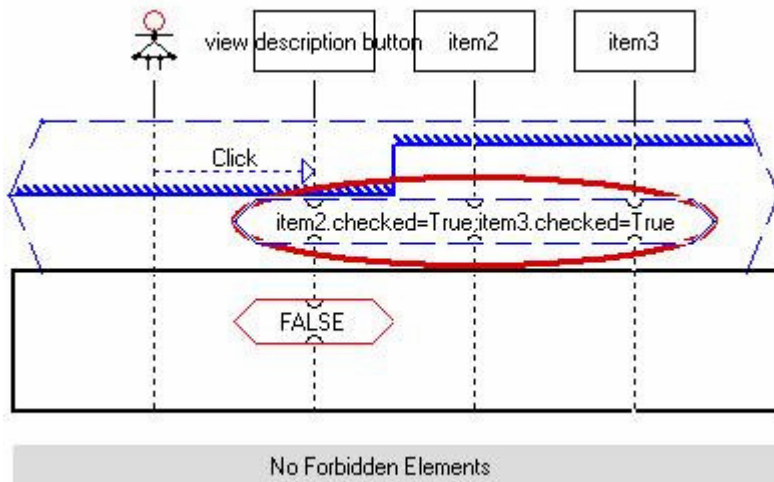


Fig. 36. Forbidden scenario 3 LSC in preactive mode, activated with a click event

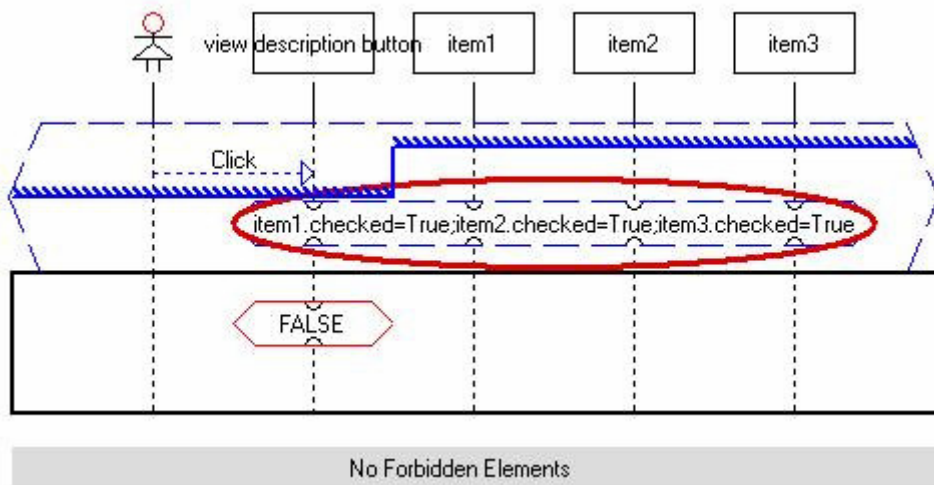


Fig. 37. Forbidden scenario 4 LSC, activated with a click event

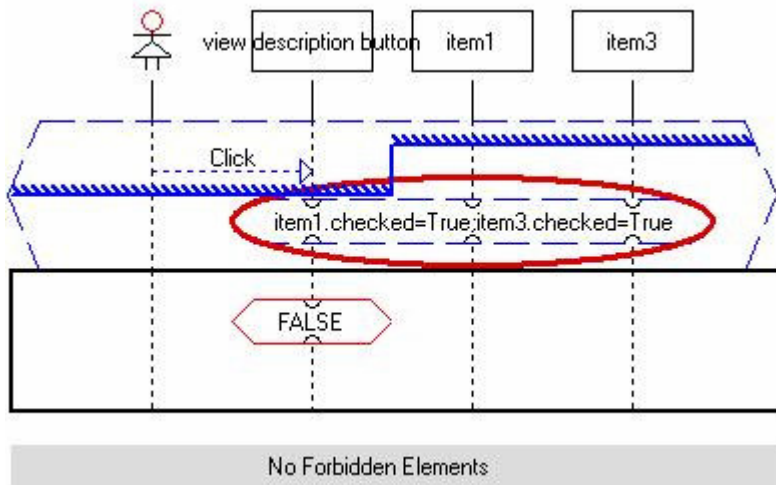


Fig. 38. View description LSC enters active mode

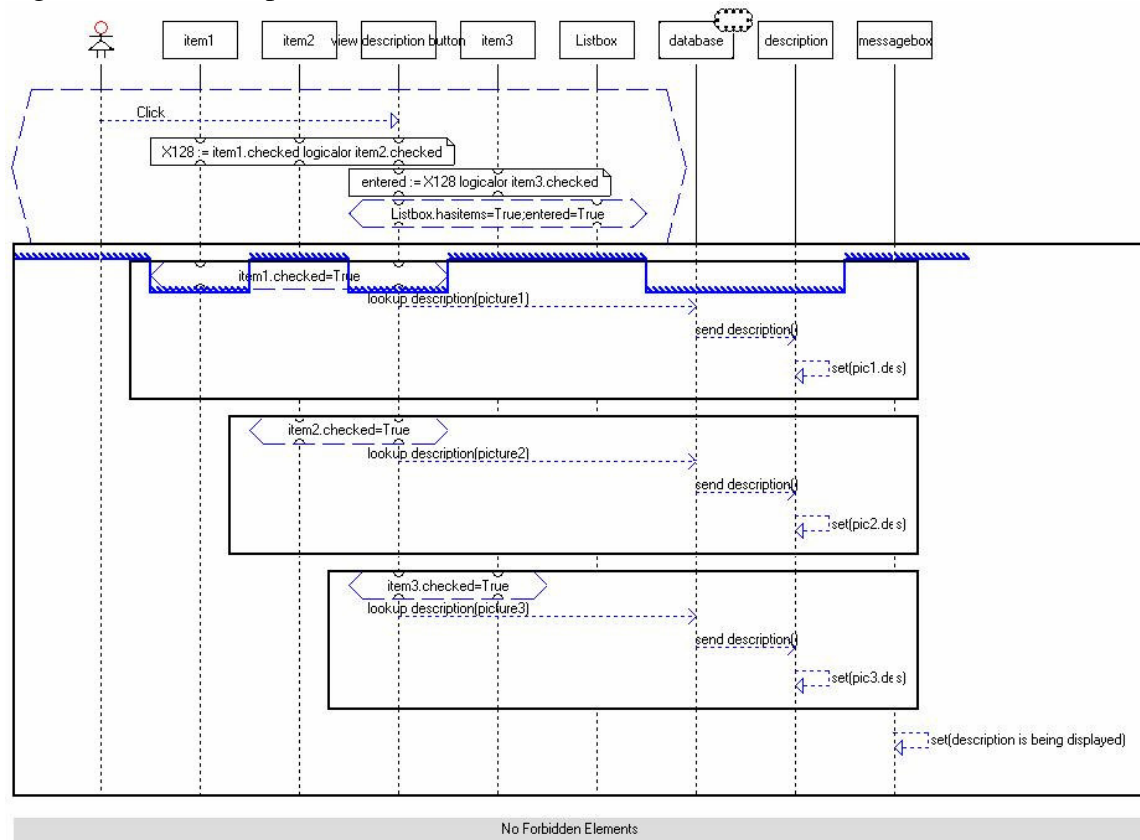


Fig. 39. Forbidden scenario 1 LSC terminates without causing an error

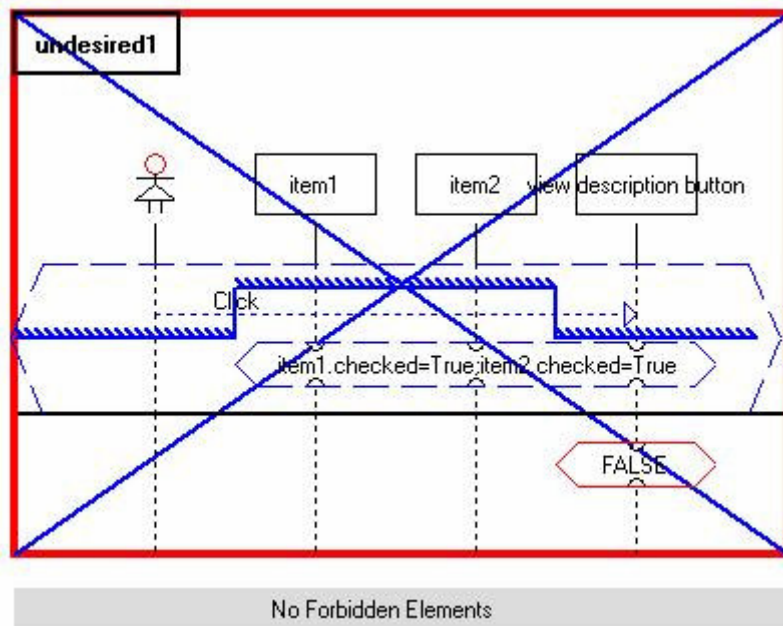


Fig. 40. Forbidden scenario 2 LSC terminates without causing an error

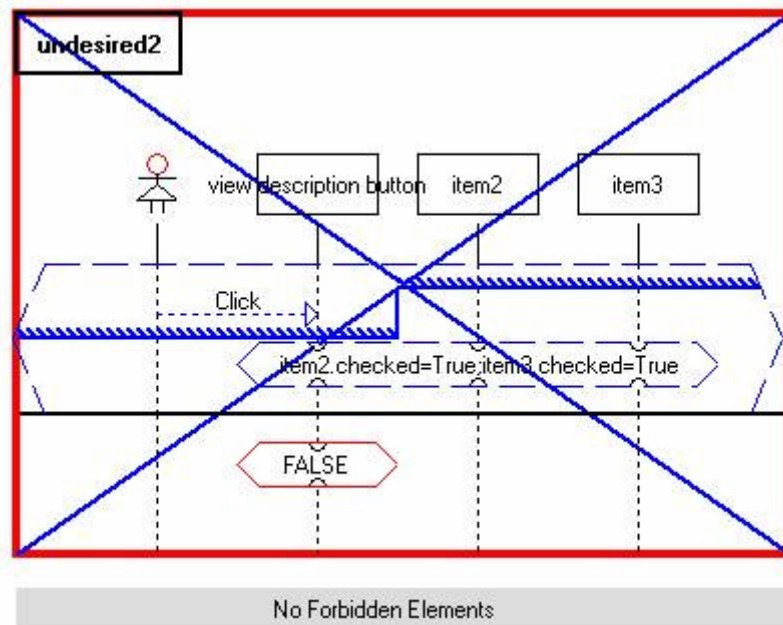


Fig. 41. Forbidden scenario 3 LSC terminates without causing an error

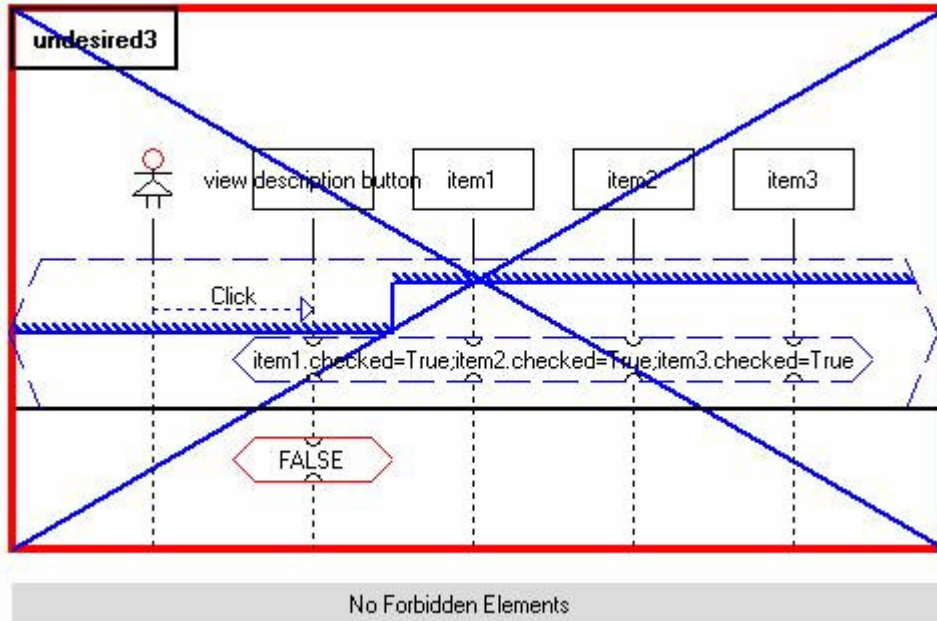
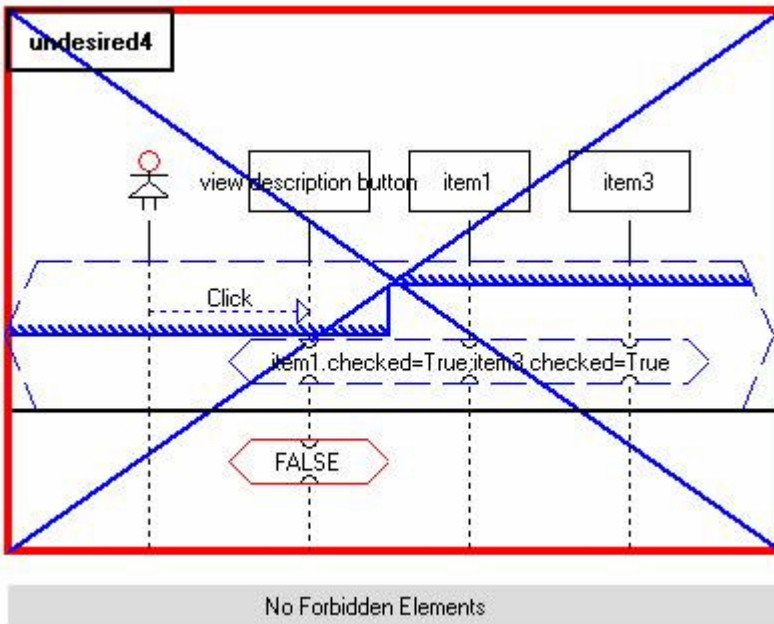


Fig. 42. Forbidden scenario 4 LSC terminates without causing an error



Now that the only active LSC is 'view descriptions', the cut advances through the events in this chart until an external event is required (figure 43). We then simulate the external event of the database sending the description to the 'description textbox' by making use of the internal objects map and the GUI. After this event, the cut advances through all the locations of the LSC so the LSC terminates and its copy is deleted (figure 44).

```

sequenceDiagram
    actor User
    participant item1
    participant item2
    participant view as view
    participant description as description
    participant item3
    participant Listbox
    participant database
    participant messageBox as messageBox

    Note over User, item1, item2, view, description, item3, Listbox, database, messageBox: Click
    Note over item1, item2: X128 := item1.checked logical or item2.checked
    Note over view, description: entered := X128 logical or item3.checked
    Note over Listbox: Listbox.hasItems=True, entered=True

    item1->>view: item1.checked=True
    view->>description: lookup_description[picture1]
    description->>database: send_description()
    database-->>messageBox: set(pic1.des)

    item2->>view: item2.checked=True
    view->>description: lookup_description[picture2]
    description->>database: send_description()
    database-->>messageBox: set(pic2.des)

    item3->>view: item3.checked=True
    view->>description: lookup_description[picture3]
    description->>database: send_description()
    database-->>messageBox: set(pic3.des)

    messageBox-->>User: set(description is being displayed)
  
```

The diagram illustrates the sequence of events in a web browser. It starts with a user clicking on a button, which triggers a series of actions. The first action is a calculation: $X128 := \text{item1.checked} \text{ logical or } \text{item2.checked}$. This is followed by another calculation: $\text{entered} := X128 \text{ logical or } \text{item3.checked}$. Then, the `Listbox` object is updated with `Listbox.hasItems=True, entered=True`. The process then enters a loop where each item (item1, item2, item3) is checked. For each item, the `checked` property is set to `True`, and a `lookup_description` message is sent to the `description` object. The `description` object then sends a `send_description()` message to the `database`, which in turn sends a `set(picX.des)` message to the `messageBox`. Finally, the `messageBox` sends a `set(description is being displayed)` message back to the user.

```

sequenceDiagram
    actor User
    participant item1
    participant item2
    participant button as view description button
    participant item3
    participant Listbox
    participant database
    participant description
    participant messagebox

    Note over User, item1, item2, button, item3, Listbox: Click
    Note over User, item1, item2, button, item3, Listbox: X128 := item1.checked logical or item2.checked
    Note over User, item1, item2, button, item3, Listbox: entered := X128 logical or item3.checked
    Note over User, item1, item2, button, item3, Listbox: Listbox.hasItems=True, entered=True

    item1->>database: look up description(picture1)
    database-->>description: end description()
    description-->>messagebox: set(pic1.des)

    item2->>database: look up description(picture2)
    database-->>description: end description()
    description-->>messagebox: set(pic2.des)

    item3->>database: look up description(picture3)
    database-->>description: end description()
    description-->>messagebox: set(pic3.des)

    messagebox-->>User: set(description is being displayed)
  
```

The diagram illustrates the 'view selected' process. It begins with a 'Click' event on the 'view description button'. A pre-condition block (dashed box) contains the following logical expressions: $X128 := \text{item1.checked} \text{ logical or } \text{item2.checked}$, $\text{entered} := X128 \text{ logical or } \text{item3.checked}$, and $\text{Listbox.hasItems} = \text{True}, \text{entered} = \text{True}$. Following this, three parallel loops are executed for 'item1', 'item2', and 'item3'. Each loop involves a 'look up description' call to the 'database', which returns 'end description()' to the 'description' object. The 'description' object then performs a 'set' action on the 'messagebox' (e.g., 'set(pic1.des)'). Finally, the 'messagebox' sends a signal back to the user: 'set(description is being displayed)'.

Note that, in this run, right after the clicking event, the cut of the `view description` LSC is advanced to the main chart (figure 38) before the LSCs for unwanted events are violated in the prechart and deleted (figures 39, 40, 41, 42). However, according to our specification, the LSCs for unwanted events might as well be deleted before we proceed with the `view description` LSC. The reason that the Play-Engine behaves this way is the execution strategy of the Play-Engine. The Play-Engine follows a maximal execution strategy, which basically means that it tries to keep a copy of an LSC alive as long as possible. The maximal execution strategy of the Play-Engine can be explained in terms of priorities of events:

- If there is an enabled event that does not violate any prechart in any LSC in a cold cut, then select it.
- If there is an enabled event that violates some prechart but does not violate any LSC in a cold cut, then select it.
- If there is an event that violates an LSC in a cold cut but does not violate any LSC in a hot cut, then select it.
- Otherwise no event can be carried out.

In our case, after the click event, the Play-Engine chooses to evaluate the cold condition in the prechart of the `view description` LSC because it is enabled and it does not violate the prechart in a cold cut. As there is no other enabled event that does not violate a prechart in the active LSCs, the Play-Engine then evaluates the cold conditions in each of the LSCs which violate their own precharts but do not violate any LSC in a cold cut.

7. Evaluation

As a reflection of our experiences using LSCs and the Play-Engine for specifying and validating requirements, now we will try to provide an evaluation of the live sequence charts language and the Play-Engine. The evaluation is done at two levels: the evaluation of the technique and the tool, and the evaluation of the specification itself. In both levels, we have made use of some frameworks provided earlier by academicians in the field.

7.1. *Method and Tool*

E.M. Clarke and J.M. Wing [27] presents the state of art in the formal methods field, and, as a conclusion, points out future directions for research and development in the field. One of the remarks they make on the future of formal methods is a set of criteria that the methods and tools should satisfy in order to be attractive to practitioners. We will now present some selected criteria that are relevant to our work, as they are given in [27], and try to evaluate our method and tool according to them.

Multiple Use: We should be able to derive benefits from a single specification at several points in a development life cycle.

Our method strongly supports multiple use. A specification depicted in terms of LSCs can be used in requirements validation, verification making use of the built-in model checker, test case generation, and synthesis of a system model from requirements. A specification given in terms of LSCs can also be used for regression testing, by employing the same test case during play-out and an actual run of the system, and comparing the results.

Integrated Use: Methods and tools should work in conjunction with each other and with common programming languages and techniques. The use of tools for formal methods should be integrated with that of tools for traditional software development.

As we have said before LSCs can be related to UML models [15]. Play-Engine interacts with Visual Basic 6.0 code for importing a GUI and communicating with the components of the GUI. Although not yet realized, Harel and Marelly [24] mentions that there is no inherent difficulty in modifying the Play-Engine so that play-in produces the formal version of the behavior in other languages than LSCs, such as temporal logic or timing diagrams. There also exists an attempt for creating state machines and statecharts from LSCs [28].

Ease of Use: Tools should be easy to use as compilers, and their output should be easy to understand.

Play-Engine was intended to be easy to use, and its interface supports this intension. However, the current version contains a lot of bugs. The tool has problems in saving and opening specifications as well as opening user interfaces and playing in or playing out

using them. While working with the tool, the user is often interrupted with runtime errors and crashes. In order to produce a specification, user should ideally save the work almost after every step and get familiar with the errors and when they occur, which might not be very easy.

Efficiency: Tools should make efficient use of a developers time.

Considering its user interface and the ideas behind it, the Play-Engine should be considered an efficient tool. However, when it comes to reality, due to the errors that we have discussed above, we should say that the Play-Engine does not make much efficient use of a developers time. As we said before, it is quite common to be interrupted with a runtime error while working, often we have to repeat some things we have done before or start all over from the beginning. Moreover, due to the problems in saving and opening files, it is possible to lose a complete specification.

Ease of Learning: Notations and tools should provide a starting point for writing formal specifications for developers who would not otherwise write them.

LSC language has a very similar notation with sequence diagrams which are widely known and used in the industry, thus it is easy to learn for most developers. Moreover having the characteristics of a graphical specification language, LSC language is much more understandable and easier to learn than mathematical specification languages. Although not much easy to use, the Play-Engine has a user friendly interface that is easy to learn just by going through the user manual.

Error Detection Oriented: Methods and tools should be optimized for finding errors, not for certifying correctness. They should support generating counterexamples as a means of debugging.

Our method for finding errors is directly executing requirements and performing test cases on the specification. Although this is a powerful technique, the methodology and the tool does not force the detection of errors inherently as the choice of test cases is completely left to the users' convenience. A means for such enforcement might be the support for generating counterexamples which is currently out of the capabilities of the Play-Engine.

Focused Analysis: Methods and tools should be good at analyzing at least one aspect of a system well. They need not be good at analyzing all aspects of a system.

The play-in/play-out methodology focuses on the reactivity of systems. The methodology is based on actions that might happen during a run and the reactions that should be taken by the system according to them. Thus, we can say that the Play-Engine supports focused analysis with a focus on reactivity.

Evolutionary Development: Methods and tools should support evolutionary system development by allowing partial specification and analysis of selected aspects of the system.

LSCs are intended to provide formalizations for use cases. When we are specifying with LSCs, we ideally relate a use case with a number of LSCs to describe it. Play-Engine can be used for evolutionary development, as we can represent each functionality of the system with a use case and then plan our iterations based on use cases. When finished with a use case we can then test and analyze the specification using play-out, without being have to develop a complete specification of the system.

Axel van Lamsweerde [26] provides a similar framework for assessing the quality of a specification technique. The criteria that are not mentioned in [27] and the evaluation of our technique according to them are as follows:

Expressive Power and Level of Coding Required: A specification language should allow properties of interest to be expressed without too much coding.

Considering our experience with LSC and Play-Engine, we should say that the LSC language has a powerful expressiveness. The only coding done in the project was two functions of one line code and a single command to add some items in the listbox. The best example proving the expressiveness of our language is the 'list pictures using keywords' use case. The LSC specification for this use case makes use of the two coded functions (logical-or and string concatenation) and is successful to produce a query string during play-out.

Communicability: The technique should be accessible for reasonably well-trained people to read high-quality specifications and check them.

As we have said before, LSC language has a very similar notation with sequence diagrams language which is very commonly known. It should not require a great deal of effort to read an LSC for someone who already knows sequence diagrams language.

7.2. Specification

In our research for criteria to be used in the quality assessment of a requirements specification, we have found out that most frameworks in the literature are designed for natural language specifications and there is a lack of frameworks to analyze requirements depicted using graphical specification languages. However, some criteria in such frameworks seem to apply our case and worth presenting, yet one must consider the advantages of a graphical specification language over natural language while going through the evaluation. The criteria below are chosen from the IEEE standard [29], given as the characteristics of a good software requirements specification.

Correct: An SRS is correct if, and only if, every requirement stated therein is one that the software shall meet.

The measure of correctness is rather difficult as there is no tool or procedure to ensure correctness, however on the account of the test cases used for validation of our specification we can say that the specification is correct.

Unambiguous: An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation.

As we have discussed before, LSC language is an exact way of specifying requirements. So our specification can be said to be unambiguous.

Complete: IEEE standard [29] states that an SRS is complete if it includes some elements, however only one of these elements is relevant to our case:

Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.

Our specification completely matches this definition, as we are mainly concerned on the reactivity of the system. The specification denotes the reactions of the system to any predictable sets of actions and inputs, for both valid and invalid cases.

Consistent: Consistency refers to internal consistency. An SRS is internally consistent if, and only if, no subset of individual requirements described in it conflict.

Assessing consistency of requirements is one of the main purposes of playing out LSCs. Our requirements have been proven to be consistent as each use case has been tested for its basic flow and alternate flows, and none of the requirements seemed to be conflicting with any other during execution.

Verifiable: An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement.

Our specification is verifiable through model checking which can be performed using the smart play-out method, an analytical strengthening of the play-out mechanism. Moreover, after having developed the system, we can verify the requirements by entering the same set of inputs during play-out and the actual execution of the software and comparing the results.

Modifiable: An SRS is modifiable if, and only if, its structure and style are such that any changes to the requirements can be made easily, completely, and consistently while retaining the structure and style.

Any specification given in terms of LSCs can be easily modified and rechecked for consistency by playing out.

Traceable: An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. The following two types of traceability are recommended:

a) *Backward traceability (i.e., to previous stages of development).*

This depends upon each requirement explicitly referencing its source in earlier documents.

b) *Forward traceability (i.e., to all documents spawned by the SRS).*

This depends upon each requirement in the SRS having a unique name or reference number.

Our specification is made up of several sets of LSCs grouped under individual use cases. The origin of the LSC specification is the use case specification document, which contains textual descriptions of each use case. The use case specification document was derived by parsing an informal user requirements document given in natural language. Our software requirements specification is easily traceable backwards to use case specification document, as both documents are organized around use cases. There exists no traceability from use case descriptions backwards to the user requirements document. However, this should not be considered as a deficiency on the account of the traceability definition given above, as an informal user requirements document written by a customer resides out of the stages of development.

8. Conclusions

Recalling our problem definition, the purpose of this research was to assess the applicability of a new requirements specification language, the live sequence charts, and its associated tool, the Play-Engine, for requirements elicitation and validation of a software project.

Prior to the assessment, we have conducted a research on theoretical issues. Theoretical studies considering the methodology and the state of art in the domain has pointed out several benefits of using live sequence charts and the Play-Engine.

The assessment was done in two levels, firstly by simply trying to capture and validate requirements with the methodology and then by evaluating the methodology according to the acquired experience.

The result of the implementation part of the project has shown that the methodology is indeed applicable as the behavioral requirements have been captured and validated successfully using play-in/play-out approach. However, we have experienced serious reliability problems with the software, during both elicitation and validation activities. The conclusion that can be drawn from the practical part of the project is that LSC specification language and play-in/play-out approach is appropriate and favorable to be used in real world applications, however the software Play-Engine has to be redeveloped or thoroughly maintained, if these ideas are to be put into industrial use.

The evaluation following the implementation was based on recommendations and quality criteria for methods and tools for specifying software systems addressed by the literature. Using these criteria, we evaluated the method and the tool, and the specification we have produced.

To conclude the evaluation of the method and the tool, we should say that the results prove that the live sequence charts language, play-in/play-out approach and the Play-Engine are promising and fit into the characteristics of the future generation of methods and tools for software specification. The drawbacks pointed out by the evaluation were mainly related with the reliability problems of the Play-Engine. The evaluation of the produced specification was of a more general perspective, primarily questioning the practical issues considering the activities involving a software requirements specification. The result of this evaluation seems to support our claim about the applicability of the methodology.

As an overall conclusion of the study, we think that this emerging method is promising and will be worthwhile to conduct future research on. Some of those future research topics that were pointed out by this study are given in the following section.

9. Future Work

As the ideas discussed and employed in this project are rather new, our research gave way to several other related research topics and also possible improvements for the current framework and the tool.

Our experiences and the evaluation addressed the main weakness about the framework as the reliability problems of the Play-Engine. The primary future effort considering the methodology should be that of redeveloping or thoroughly maintaining the Play-Engine, as the reliability problems not only stop people from using the software but also form a bottleneck for understanding and analyzing the theory behind.

We have also addressed that the Play-Engine might also be improved, in terms of both incorporating new features and improving the current features. The possible future improvements to current features were pointed out as a result of the practical work with the Play-Engine. On the other hand, the new features that might be included into Play-Engine were pointed out during literature research and evaluation.

The very first improvement issue we have addressed was about types. In the beginning of the project when we were trying to create our own types for the GUI objects such as a list box or a combo box, we experienced a lack of support for defining new types and finally were bound to using the predefined types given in the Play-Engine libraries. The support for defining types should absolutely be emphasized in the case of a redevelopment or maintenance of the Play-Engine. In addition, the types for the most popular GUI objects might be defined in advance and included in a type library, which will decrease complications for the user. Another improvement to the Play-Engine, inspired by the internal objects map, is to extend this feature into an object model diagram, which will support associations, composition and inheritance between objects. If this is accomplished, the integration of our method with UML models as well as object oriented programming paradigms will be supported much better. Furthermore, the Play-Engine can be improved so that it supports creation and destruction of objects, thus we do not have to work with a fixed number of objects. Description of future work considering object oriented concepts is given in more detail in [24].

While evaluating the Play-Engine as a tool we have noticed a very nice new feature complementary to the current mechanisms which is the generation of counter examples. As we have discussed before, in Play-Engine, the validation of requirements are done by playing out a set of test cases during play-out. However the validation process is a soft one because it depends on the test cases picked up by the user. An interesting research issue might be the development of a tool which can generate test cases that will invalidate a specification given as a set of LSCs. This tool can then be integrated into Play-Engine. Another useful feature that can be included into the Play-Engine is the means for saving the specification in different specification languages. This will provide the support for checking the specifications with each other, and using different validation/verification techniques available for a number of languages on a single specification. Besides the issues about validation/verification, the Play-Engine can be provided with the means for

generating simple program code from the specification. As a specification is the basis for the implementation, generation of a skeleton code of the system directly from its specification will decrease misunderstandings and inconsistencies between the specification and the developed system dramatically.

In this research and the literature on LSCs and the Play-Engine, the main consideration was the software development activities. However, today there exist many software projects on maintenance or redevelopment of legacy systems which are sometimes more complicated processes. With the motivation of using the presented ideas also for such activities, we think that the development of a tool which generates LSC specifications from a given program code might be an interesting future research topic.

Finally, in addition to improving the methodology and its uses, we think that further research should be conducted with the aim of providing a wider acceptance of these ideas. Recalling from chapter 4, the primary motivation of LSCs and the Play-Engine was providing the means for simulating system behavior and direct testing of requirements with inter-object models, instead of intra-object models which cost more. A comparative study, based on costs and capabilities of using intra-object models and the proposed approach to inter-object models for model execution and requirements validation can be conducted to further investigate the benefits of the methodology. Other than trying to prove the benefits, future research can demonstrate the applicability of the ideas by composing them with popular development frameworks. Such an attempt might be integrating the ideas with the requirements and test disciplines of RUP and restructuring the workflows of these disciplines.

References

- [1]G. Booch, I. Jacobson, J. Rumbaugh. Managing Software Requirements, A Use Case Approach. Second edition, 2003.
- [2]C. Jones. "Revitalizing Software Project Management." American Programmer 6(7), pp. 3-12.1994.
- [3]A. M. Davis. Software Requirements: Objects, Functions, and States. Englewood Cliffs, NJ: Prentice-Hall. 1993.
- [4]K. Pohl. "The Three Dimensions of Requirements Engineering," in Proceedings of Fifth International Conference on Advanced Information Systems Engineering (CAiSE '93), Paris, pp. 275-292.1993.
- [5]K. L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Application. IEEE Trans. Software Eng. 6(1), pp. 2-13.1980.
- [6]S. Flake, W. Mueller, and J. Ruf. Structured English for model checking specifications. In Proc. of the GI-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", Frankfurt, Germany, February 2000.
- [7]Rapide Design Team. The Rapide-1 Executable Language Reference Manual, Program Analysis and Verification Group, Computer Systems Lab., Stanford University, version 1 edition, October 1994.
- [8]J.M. Spivey. The Z Notation: A Reference Manual, 2nd edition., London Prentice-Hall, 1992.
- [9]J.R. Abrial. The B Book - Assigning Programs to Meanings. Cambridge University Press, August 1996.
- [10]T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. Computer Networks, 14 (1), 25-59, 1987.
- [11]C.B. Jones. Systematic Software Development Using VDM. Prentice-Hall International, New York, 1986.
- [12]M. Ramachandran. Requirements-Driven Software Test: A Process-Oriented Approach, 1996.
- [13]M.R. Blackburn, Software Productivity Consortium, Herndon, Virginia, R. D. Busser, Motorola, Boynton Beach, Florida. Requirements for Industrial-Strength Formal Method Tools,1998.

- [14]J. Klose. Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior. PhD thesis, Carl von Ossietzky, Universitat Oldenburg, 2003.
- [15]J. Klose and B. Westphal. Relating LSC Specifications to UML models, Universitat Oldenburg, 2002.
- [16]Play-Engine User's Guide.
- [17]ITU-T: ITU-T Recommendation Z.120: Message Sequence Chart (MSC). ITU-T, Geneva, 1993.
- [18]D. Ross and K. Schoman. "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, 3, 1, 1977.
- [19]ITU-T Recommendation Z.100 Annex F(11/00), SDL Formal Semantics Definition, International Telecommunication Union, Geneva, 2001.
- [20]I. Jacobson. Object-Oriented Software Engineering, A Use Case Driven Approach. Addison-Wesley, ACM Press 1993.
- [21]R. Pooley and P. Stevens. Using UML: Software Engineering with Objects and Components. Addison Wesley, 1999.
- [22]J. L. Peterson. Petri Net Theory and the Modeling of Systems, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
- [23]J. Rumbaugh, I. Jacobson, G. Booch. The Unified Modeling Language Reference Manual, 2nd Edition, Addison-Wesley Object Technology Series, 2004.
- [24]D. Harel and R. Marelly. Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine, Springer-Verlag, 2003.
- [25]IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.121990 (Revision and redesignation of IEEE std 7921983).
- [26]A. V. Lamsweerde. Formal Specification: a Roadmap, International Conference on Software Engineering, Proceedings of the Conference on The Future of Software Engineering, Limerick, Ireland, 2000.
- [27]E.M. Clarke and J.M. Wing. Formal methods: state of the art and future directions. ACM Computing Surveys 28(4), 626-643, 1996.
- [28]D. Harel and H. Kugler. Synthesizing State-Based Object Systems from LSC Specifications. Int. J. of Foundations of Computer Science (IJFCS), 13(1):5-51, February 2002.

[29]IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830-1998 (Revision of IEEE Std 830-1993).

Appendix A Questionnaire for Requirements Elicitation

Part I: Establishing the Customer or User Profile

Name:

Company:

Industry:

Job title:

1. What are your key responsibilities?

.....

.....

.....

2. How is success measured?

.....

.....

.....

3. Which problems interfere with your success?

.....

.....

.....

4. What, if any, trends make your job easier or more difficult?

.....

.....

.....

Part II: Understanding the User Environment

1. Who are the users?

.....

.....

2. What is their educational background?

.....

.....

3. What is their computer background?

.....

.....

4. Are users experienced with this type of application?

.....

.....

5. What platforms are in use?

.....

.....

Part III: Recap for Understanding

You have mentioned the following problems in your request:

- Indexing pictures by means of keywords of your choice
- Listing the pictures by searching the archive with a keyword
- Viewing the selected pictures from the list
- Attaching descriptions to each picture
- Viewing the description for a selected picture
- Printing out the picture
- Printing out the description
- Recording the archive in some format and being able to load it later in another machine with the software installed

Does this represent the problems you are having with your existing solution?

.....
.....
.....

What, if any, other problems you are experiencing?

.....
.....
.....

Part IV: Analyst's Inputs on Stakeholder's Problem (validate or invalidate assumptions)

1. About indexing and retrieving the pictures from the archive with keywords:

You have explained this need by providing some examples in your description. Taking these as basis we are assuming that these keywords will belong to some sort of category, such as location or region, time period, architectural style, name. *Is this assumption correct?*

.....
If it is correct, what are the categories that a keyword might belong?
.....
.....
.....

You have also mentioned that a picture might be an architectural construct or a drawing. *Do these categories apply to drawings?*

2. About viewing descriptions and pictures:

One of your needs is to view the pictures and their descriptions. We are assuming that after getting the pictures listed you will select them and view. *Do you want to select and view multiple pictures or just one at a time?*

.....
.....
If you want to view multiple pictures, how would you like them to be presented? Some possible ways of presenting them might be slide show, thumbnails (pictures with smaller size).
.....
.....

You have also mentioned that you want to view descriptions of pictures. *Do you want the descriptions to be displayed each time the picture is displayed or do you want them to be displayed on your command?*
.....
.....

3. About printing out pictures and descriptions:

Do you want to print just one picture at a time or print multiple pictures by selecting them from the listed pictures?
.....
.....

Part V: Assessing the Opportunity

1. Who needs this application in your organization?

.....
.....

2. How many of these types of users would use the application?

.....
.....

Part VI: Analyst's Summary

[to be filled in by requirements analyst, Alper Ersoy]

- 1.
- 2.
- 3.

TRITA-CSC-E 2006:037
ISRN-KTH/CSC/E--06/037--SE
ISSN-1653-5715